Chapter 3

Deterministic planning

In this chapter we describe a number of algorithms for solving the historically most important and most basic type of planning problem. Two rather strong simplifying assumptions are made. First, all actions are deterministic, that is, under every action every state has at most one successor state. Second, there is only one initial state.

Under these restrictions, whenever a goal state can be reached, it can be reached by a fixed sequence of actions. With more than one initial state it would be necessary to use a different sequence of actions for every initial state, and with nondeterministic actions the sequence of actions to be taken is not simply a function of the initial state, and for producing appropriate sequences of actions a more general notion of plans with branches/conditionals becomes necessary. This is because after executing an action, even when the starting state was known, the state that is reached cannot be predicted, and the way plan execution continues depends on the new state. In Chapter 4 we relax both of these restrictions, and consider planning with more than one initial state and with nondeterministic actions.

The structure of this chapter is as follows. First we discuss the two ways of traversing the transition graphs without producing the graphs explicitly. In forward traversal we repeatedly compute the successor states of our current state, starting from the initial state. In backward traversal we must use sets of states, represented as formulae, because we must start from the set of goal states, and further, under a given action a state may have several predecessor states.

Then we discuss the use of heuristic search algorithms for performing the search in the transition graphs and the computation of distance heuristics to be used in estimating the value of the current states or sets of states. Further improvements to plan search are obtained by recognizing symmetries in the transition graphs, and for backward search, restricting the search by invariants that are formulae describing which states are reachable from the initial state.

A complementary approach to planning is obtained by translating the planning problem to the classical propositional logic and then finding plans by algorithms that test the satisfiability of formulae in the propositional logic. This is called satisfiability planning. We discuss two translations of deterministic planning to the propositional logic. The second translation is more complicated but also more efficient as it avoids considering all interleavings of a set of mutually independent operators.

We conclude the chapter by presenting the main results on the computational complexity of deterministic planning.

3.1 Problem definition

We formally define the deterministic planning problem.

Definition 3.1 *A 4-tuple $\langle A, I, O, G \rangle$ consisting of a set A of state variables, a state I (a valuation of A), a set O of operators over A, and a propositional formula G over A, is* a problem instance in deterministic planning.

The state I is the *initial state* and the formula G describes the set of *goal states*.

Definition 3.2 *Let $\Pi = \langle A, I, O, G \rangle$ be a problem instance in deterministic planning. A sequence $o_1, \ldots, o_n$ of operators is a* plan *for Π if and only if $app_{o_n}(app_{o_{n-1}}(\cdots app_{o_1}(I) \cdots)) \models G$, that is, when applying the operators $o_1, \ldots, o_n$ in this order starting in the initial state, one of the goal states is reached.*

3.2 State-space search

The simplest planning algorithm just generates all states (valuations of A), constructs the transition graph, and then finds a path from the initial state I to a goal state $g \in G$ for example by a shortest-path algorithm. The plan is then simply the sequence of actions corresponding to the edges on the shortest path from the initial state to a goal state.

However, this algorithm is in general not feasible when the number of state variables is higher than 20 or 30, as the number of valuations is very high: $2^{20} = 1048576 \sim 10^6$ for 20 Boolean state variables, and $2^{30} = 1073741824 \sim 10^9$ for 30.

Instead, it will often be much more efficient to avoid generating most of the state space explicitly, and just to produce the successor or predecessor states of the states currently under consideration. This is how many of the modern planning algorithms work.

There are two main possibilities in finding a path from the initial state to a goal state: traverse the transition graph forward starting from the initial state, or traverse it backwards starting from the goal states.

The main difference between these is caused by the fact that there may be several goal states (and even one goal state may have several possible predecessor states with respect to one operator) but only one initial state: in forward traversal we repeatedly compute the unique successor state of the current state, whereas with backward traversal we are forced to keep track of a possibly very high number of possible predecessor states of the goal states.

Again, it is difficult to say which one is in general better. Backward search is slightly more complicated to implement, but when the number of goal states is high, it allows to simultaneously consider a high number of potential suffixes of a plan, each leading to one of the goal states.

3.2.1 Progression and forward search

We already defined progression for single states s as $app_o(s)$, and the definition of the deterministic planning problem in Section 3.1 suggests a simple algorithm that does not require the explicit representation of the transition graph: generate a search tree starting from the initial state as the root node, and generate the children nodes by computing successor states by progression. Any node corresponding to a state s such that $s \models G$ corresponds to a plan: the plan is simply the sequence of operators from the root node to the node.

Later in this chapter we discuss more sophisticated ways of doing plan search with progression, as well as computation of distance estimates for guiding heuristic search algorithms.

3.2.2 Regression and backward search

With backward search the starting point is a propositional formula G that describes the set of goal states. An operator is selected, and the set of possible predecessor states is computed, and this set again is described by a propositional formula. One step in this computation, called *regression*, is more complicated than computing unique successor states of deterministic operators by progression. Reasons for this are that a state and an operator do not in general determine the predecessor state uniquely (one state may have several predecessors), and that we have to handle arbitrary propositional formulae instead of single states.

Definition 3.3 *We define the condition $EPC_l(o)$ of literal l becoming true when the operator $\langle c, e \rangle$ is applied as $EPC_l(e)$ defined recursively as follows.*

$$
\begin{aligned}
EPC_l(\top) &= \bot \\
EPC_l(l) &= \top \\
EPC_l(l') &= \bot \text{ when } l \neq l' \text{ (for literals } l') \\
EPC_l(e_1 \wedge \cdots \wedge e_n) &= EPC_l(e_1) \vee \cdots \vee EPC_l(e_n) \\
EPC_l(c \rhd e) &= EPC_l(e) \wedge c
\end{aligned}
$$

For effects e, the truth-value of the formula $EPC_l(e)$ indicates whether l is one of the literals that the effect e assigns the value true. The connection to the earlier definition of $[e]_s$ is explained by the following lemma.

Lemma 3.4 *Let A be the set of state variables, s be a state on A, l a literal on A, and e an effect on A. Then $l \in [e]_s$ if and only if $s \models EPC_l(e)$.*

Proof: Proof is by induction on the structure of the effect e.

Base case 1, $e = \top$: By definition of $[\top]_s$ we have $l \notin [\top]_s = \emptyset$, and by definition of $EPC_l(\top)$ we have $s \not\models EPC_l(\top) = \bot$, so the equivalence holds.

Base case 2, $e = l$: $l \in [l]_s = \{l\}$ by definition, and $s \models EPC_l(l) = \top$ by definition.

Base case 3, $e = l'$ for some literal $l' \neq l$: $l \notin [l']_s = \{'l\}$ by definition, and $s \not\models EPC_l(l') = \bot$ by definition.

Inductive case 1, $e = e_1 \wedge \cdots \wedge e_n$:

$$
\begin{aligned}
l \in [e]_s \quad &\text{if and only if } l \in [e']_s \text{ for some } e' \in \{e_1, \ldots, e_n\} \\
&\text{if and only if } s \models EPC_l(e') \text{ for some } e' \in \{e_1, \ldots, e_n\} \\
&\text{if and only if } s \models EPC_l(e_1) \vee \cdots \vee EPC_l(e_n) \\
&\text{if and only if } s \models EPC_l(e_1 \wedge \cdots \wedge e_n).
\end{aligned}
$$

The second equivalence is by the induction hypothesis, the other equivalences are by the definitions of $EPC_l(e)$ and $[e]_s$ as well as elementary facts about propositional formulae.

Inductive case 2, $e = c \rhd e'$:

$$
\begin{aligned}
l \in [c \rhd e']_s \quad &\text{if and only if } l \in [e']_s \text{ and } s \models c \\
&\text{if and only if } s \models EPC_l(e') \text{ and } s \models c \\
&\text{if and only if } s \models EPC_l(c \rhd e').
\end{aligned}
$$

The second equivalence is by the induction hypothesis.

This completes the proof.   □

Notice that any operator $\langle c, e \rangle$ can be expressed in normal form in terms of $EPC_a(e)$ as

$$\left\langle c, \bigwedge_{a \in A} (EPC_a(e) \rhd a) \wedge (EPC_{\neg a}(e) \rhd \neg a) \right\rangle .$$

The formula $(a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$ expresses the truth-value of $a \in A$ after applying o in terms of truth-values of formulae before applying o: either a was true before and did not become false, or a became true.

Lemma 3.5 *Let $a \in A$ be a state variable and $o = \langle c, e \rangle \in O$ an operator. Let s be a state and $s' = app_o(s)$. Then $s \models (a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$ if and only if $s' \models a$.*

Proof: Assume that $s \models (a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$. We perform a case analysis and show that $s' \models a$ holds in both cases.

Case 1: Assume that $s \models a \wedge \neg EPC_{\neg a}(e)$. By Lemma 3.4 $\neg a \notin [e]_s$. Hence a remains true in s'.

Case 2: Assume that $s \models EPC_a(e)$. By Lemma 3.4 $a \in [e]_s$, and hence $s' \models a$.

For the other half of the equivalence, assume that $s \not\models (a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$. Hence $s \models (\neg a \vee EPC_{\neg a}(e)) \wedge \neg EPC_a(e)$.

Assume that $s \models a$. Now $s \models EPC_{\neg a}(e)$ because $s \models \neg a \vee EPC_{\neg a}(e)$, and hence by Lemma 3.4 $\neg a \in [e]_s$ and hence $s' \not\models a$.

Assume that $s \not\models a$. Because $s \models \neg EPC_a(e)$, by Lemma 3.4 $a \notin [e]_s$ and hence $s' \not\models a$.

Therefore $s' \not\models a$ in all cases.   □

The formulae $EPC_l(o)$ can now be used in defining regression for operators o.

Definition 3.6 (Regression) *Let ϕ be a propositional formula. Let $\langle p, e \rangle$ be an operator. The regression of ϕ with respect to $o = \langle p, e \rangle$ is $regr_o(\phi) = \phi_r \wedge p \wedge f$ where ϕ_r is obtained from ϕ by replacing every proposition $a \in A$ by $(a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$, and $f = \bigwedge_{a \in A} \neg(EPC_a(e) \wedge EPC_{\neg a}(e))$. We also define $regr_e(\phi) = \phi_r \wedge f$.*

The conjuncts of f say that none of the state variables may simultaneously become true and false.

Because $regr_e(\phi)$ often contains many occurrences of $\bot$ and $\top$, it is useful to simplify it by applying equivalences like $\top \wedge \phi \equiv \phi$, $\bot \wedge \phi \equiv \bot$, $\top \vee \phi \equiv \top$, $\bot \vee \phi \equiv \phi$, $\neg \bot \equiv \top$, and $\neg \top \equiv \bot$.

Regression can equivalently be defined in terms of the conditions the state variables stay or become false, that is, we could use the formula $(\neg a \wedge \neg EPC_a(e)) \vee EPC_{\neg a}(e)$ which tells when a is false. The negation of this formula, which can be written as $(a \wedge \neg EPC_{\neg a}(e)) \vee (EPC_a(e) \wedge \neg EPC_{\neg a}(e))$, is not equivalent to $(a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$. However, if $EPC_a(e)$ and $EPC_{\neg a}(e)$ are never simultaneously true, we do get equivalence, that is,

$$\neg(EPC_a(e) \wedge EPC_{\neg a}(e)) \quad \models \quad \begin{aligned} &((a \wedge \neg EPC_{\neg a}(e)) \vee (EPC_a(e) \wedge \neg EPC_{\neg a}(e))) \\ &\leftrightarrow ((a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)) \end{aligned}$$

because $\neg(EPC_a(e) \wedge EPC_{\neg a}(e)) \models (EPC_a(e) \wedge \neg EPC_{\neg a}(e)) \leftrightarrow EPC_a(e)$.

Concerning the worst-case size of the formula obtained by regression with operators $o_1, \ldots, o_n$ starting from ϕ, the obvious upper bound on its size is the product of the sizes of $\phi, o_1, \ldots, o_n$, which is exponential in n. However, because of the many possibilities of simplifying the formulae and the typically simple structure of the operators, the formulae can often be simplified a lot. For unconditional operators $o_1, \ldots, o_n$ (with no occurrences of $\rhd$), an upper bound on the size of the formula (after the obvious simplifications that eliminate occurrences of $\top$ and $\bot$) is the sum of the sizes of $o_1, \ldots, o_n$ and ϕ.

The reason why regression is useful for planning is that it allows to compute the predecessor states by simple formula manipulation. The same is not possible for progression, that is, there does not seem to be a simple definition of successor states of a *set* of states expressed in terms of a formula: simple syntactic progression is restricted to individual states only.

The important property of regression is formalized in the following lemma.

Lemma 3.7 *Let ϕ be a formula over A. Let o be an operator with effect e. Let s be any state and $s' = app_o(s)$. Then $s \models regr_e(\phi)$ if and only if $s' \models \phi$.*

Proof: The proof is by structural induction over subformulae ϕ' of ϕ. We show that the formula ϕ_r obtained from ϕ by replacing propositions $a \in A$ by $(a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$ has the same truth-value in s as ϕ has in s'.

Induction hypothesis: $s \models \phi'_r$ if and only if $s' \models \phi'$.

Base case 1, $\phi' = \top$: Now $\phi'_r = \top$ and both are true in the respective states.

Base case 2, $\phi' = \bot$: Now $\phi'_r = \bot$ and both are false in the respective states.

Base case 3, $\phi' = a$ for some $a \in A$: Now $\phi'_r = (a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$. By Lemma 3.5 $s \models \phi'_r$ if and only if $s' \models \phi'$.

Inductive case 1, $\phi' = \neg\psi$: By the induction hypothesis $s \models \psi_r$ iff $s' \models \psi$. Hence $s \models \phi'_r$ iff $s' \models \phi'$ by the truth-definition of $\neg$.

Inductive case 2, $\phi' = \psi \vee \psi'$: By the induction hypothesis $s \models \psi_r$ iff $s' \models \psi$, and $s \models \psi'_r$ iff $s' \models \psi'$. Hence $s \models \phi'_r$ iff $s' \models \phi'$ by the truth-definition of $\vee$.

Inductive case 3, $\phi' = \psi \wedge \psi'$: By the induction hypothesis $s \models \psi_r$ iff $s' \models \psi$, and $s \models \psi'_r$ iff $s' \models \psi'$. Hence $s \models \phi'_r$ iff $s' \models \phi'$ by the truth-definition of $\wedge$. $\square$

Operators for regression can be selected arbitrarily, but there is a simple property all useful regression steps satisfy. For example, regressing a with the effect $\neg a$ is not useful, because the new formula $\bot$ describes the empty set of states, and therefore the operators leading to it from the goal formula are not the suffix of any plan. Another example is regressing a with the operator $\langle b, c \rangle$, yielding $regr_{\langle b,c\rangle}(a) = a \wedge b$, which means that the set of states becomes smaller. This does not rule out finding a plan, but finding a plan is more difficult than it was before the regression step, because the set of possible prefixes of a plan leading to the current set of states is smaller than it was before. Hence it would be better not to take this regression step.

Lemma 3.8 *Let there be a plan $o_1, \ldots, o_n$ for $\langle A, I, O, G \rangle$. If $regr_{o_k}(regr_{o_{k+1}}(\cdots regr_{o_n}(G) \cdots)) \models regr_{o_{k+1}}(\cdots regr_{o_n}(G) \cdots)$ for some $k \in \{1, \ldots, n-1\}$, then also $o_1, \ldots, o_{k-1}, o_{k+1}, \cdots, o_n$ is a plan for $\langle A, I, O, G \rangle$.*

Proof: $\square$

Hence any regression step that makes the set of states smaller in the set-inclusion sense is unnecessary. However, testing whether this is the case may be computationally expensive.

Lemma 3.9 *The problem of testing that $regr_o(\phi) \not\models \phi$ is NP-hard.*

Proof: We give a reduction from the NP-complete satisfiability problem of the propositional logic.

Let ϕ be any formula. Let a be a propositional variable not occurring in ϕ. Now $regr_{\langle\neg\phi\rightarrow a,a\rangle}(a) \not\models a$ if and only if $(\neg\phi \rightarrow a) \not\models a$, because $regr_{\langle\neg\phi\rightarrow a,a\rangle}(a) = \neg\phi \rightarrow a$. $(\neg\phi \rightarrow a) \not\models a$ is equivalent to $\not\models (\neg\phi \rightarrow a) \rightarrow a$ that is equivalent to the satisfiability of $\neg((\neg\phi \rightarrow a) \rightarrow a)$. Further, $\neg((\neg\phi\rightarrow a)\rightarrow a)$ is logically equivalent to $\neg(\neg(\phi \vee a) \vee a)$ and further to $\neg(\neg\phi \vee a)$ and $\phi \wedge \neg a$.

Satisfiability of $\phi \wedge \neg a$ is equivalent to the satisfiability of ϕ as a does not occur in ϕ: if ϕ is satisfiable, there is a valuation v such that $v \models \phi$, we can set a false in v to obtain v', and as a does not occur in ϕ, we still have $v' \models \phi$, and further $v' \models \phi \wedge \neg a$. Clearly, if ϕ is unsatisfiable also $\phi \wedge \neg a$ is.

Hence $regr_{\langle\neg\phi\rightarrow a,a\rangle}(a) \not\models a$ if and only if ϕ is satisfiable. $\qquad\square$

The problem is also in NP, but we do not show it here. Also the following problem is in NP, but we just show the NP-hardness. The question is whether an empty set of states is produced by a regression step, that is, whether the resulting formula is unsatisfiable.

Lemma 3.10 *The problem of testing that $regr_o(\phi)$ is satisfiable is NP-hard.*

Proof: By a reduction from satisfiability in the propositional logic. Let ϕ be a formula. $regr_{\langle\phi,a\rangle}(a)$ is satisfiable if and only if ϕ is satisfiable because $regr_{\langle\phi,a\rangle}(a) \equiv \phi$.

The problem is NP-hard even if we restrict to operators that have a satisfiable precondition: ϕ is satisfiable if and only if $(\phi \vee \neg a) \wedge a$ is satisfiable if and only if $regr_{\langle\phi\vee\neg a,b\rangle}(a \wedge b)$ is satisfiable. Here a is a proposition not occurring in ϕ. Clearly, $\phi \vee \neg a$ is true when a is false, and hence $\phi \vee \neg a$ is satisfiable. $\qquad\square$

Of course, testing that $regr_o(\phi) \not\models \phi$ or that $regr_o(\phi)$ is satisfiable is not necessary for the correctness of backward search, but avoiding useless steps improves its efficiency.

Early work on planning restricted to goals and operator preconditions that are conjunctions of state variables, and to unconditional operator effects (STRIPS operators.) In this special case both goals G and operator effects e can be viewed as sets of literals, and the definition of regression is particularly simple: regressing G with respect to $\langle c,e \rangle$ is $(G\backslash e) \cup c$. If there is $a \in A$ such that $a \in G$ and $\neg a \in e$, then the result of regression is $\bot$, that is, the empty set of states. We do not use this restricted type of regression in this lecture.

Some planners that use backward search and have operators with disjunctive preconditions and conditional effects eliminate all disjunctivity by branching: for example, the backward step from g with operator $\langle a \vee b, g \rangle$, producing $a \vee b$, is handled by producing two branches in the search tree, one for a and another for b. Disjunctivity caused by conditional effects can similarly be handled by branching. However, this branching leads to a very high branching factor for the search tree and thus to poor performance.

In addition to being the basis of backward search, regression has many other useful applications in reasoning about actions and formal manipulation of operators.

Definition 3.11 (Composition of operators) *Let $o_1 = \langle p_1, e_1 \rangle$ and $o_2 = \langle p_2, e_2 \rangle$ be two operators on A. Then their* composition $o_1 \circ o_2$ *is defined as*

$$\left\langle p, \bigwedge_{a \in A} \left(\begin{array}{c} ((regr_{e_1}(EPC_a(e_2))) \vee (EPC_a(e_1) \wedge \neg regr_{e_1}(EPC_{\neg a}(e_2)))) \rhd a) \wedge \\ ((regr_{e_1}(EPC_{\neg a}(e_2))) \vee (EPC_{\neg a}(e_1) \wedge \neg regr_{e_1}(EPC_a(e_2)))) \rhd \neg a) \end{array} \right) \right\rangle$$

where $p = p_1 \wedge regr_{e_1}(p_2) \wedge \bigwedge_{a \in A} \neg (EPC_a(e_1) \wedge EPC_{\neg a}(e_1))$.

Notice that in $o_1 \circ o_2$ first is o_1 is applied and then o_2, so the ordering is opposite to the usual notation for the composition of functions.

Theorem 3.12 *Let o_1 and o_2 be operators and s a state. Then $app_{o_1 \circ o_2}(s)$ is defined if and only if $app_{o_1;o_2}(s)$ is defined, and $app_{o_1 \circ o_2}(s) = app_{o_1;o_2}(s)$.*

Proof: □

The above construction can be used in eliminating *sequantial composition* from operator effects (Section 2.3.1).

3.3 Planning by heuristic search algorithms

Plan search can be performed in the forward or in the backward direction respectively with progression or regression, as described in Sections 3.2.1 and 3.2.2. There are several obvious algorithms that could be used for the purpose, including depth-first search, breadth-first search and iterative deepening, but without informed selection of branches of search trees these algorithms perform poorly.

The use of additional information for guiding search is essential for achieving efficient planning with general-purpose search algorithms. Algorithms that use heuristic estimates on the values of the nodes in the search space for guiding the search have been applied to planning very successfully. Some of the more sophisticated search algorithms that can be used are A∗ [Hart *et al.*, 1968], WA∗ [Pearl, 1984], IDA∗ [Korf, 1985], simulated annealing [Kirkpatrick *et al.*, 1983].

The effectiveness of these algorithms is dependent on good heuristics for guiding the search. For planning with progression and regression the main heuristic information is in the form of estimates on the distance between states. The distance is the minimum number of operators needed for reaching a state from another state. In Section 3.4 we present techniques for estimating the distances between states and sets of sets. In this section we discuss how heuristic search algorithms are applied in planning assuming that we have a useful heuristics for guiding these algorithms

When plan search proceeds by progression in forward direction starting from the initial state, we estimate the distance between the current state and the set of goal states. When plan search proceeds by regression in backward direction starting from the goal states, we estimate the distance between the initial state and the current set of goal states as computed by regression.

For progression, the search tree nodes are sequences of operators (prefixes of plans.)

$$o_1, o_2, \ldots, o_n$$

The initial node for search is the empty sequence. The children nodes are obtained by progression with respect to an operator or by dropping out some of the last operators.

Definition 3.13 (Children for progression) *Let $\langle A, I, O, G \rangle$ be a problem instance. For progression, the children of a search tree node $o_1, o_2, \ldots, o_n$ are the following.*

    1. $o_1, o_2, \ldots, o_n, o$ *for any $o \in O$ such that $app_{o_1;\ldots;o_n;o}(I)$ is defined*

    2. $o_1, o_2, \ldots, o_i$ *for any $i < n$*

When $app_{o_1;o_2\ldots,o_n}(I) \models G$ then $o_1, \ldots, o_n$ is a plan.

For regression, the nodes of the search tree are also sequences of operators (suffixes of plans.)

$$o_n, \ldots, o_1$$

The initial node for search is the empty sequence. The children of a node are those obtained by prefixing the current sequence with an operator or by dropping out some of the first actions and associated formulae.

Definition 3.14 (Children for regression) *Let $\langle A, I, O, G \rangle$ be a problem instance. For regression, the children of node $\phi_n, \ldots, o_1$ are the following.*

    1. $o, o_n, \ldots, o_1$ *for any $o \in O$*

    2. $o_i, \ldots, o_1$ *for any $i < n$*

When $I \models regr_{o_n;\ldots;o_1}(G)$ the sequence $o_n, \ldots, o_1$ is a plan.

For both progression and regression the neighbors that are obtained by removing some operators from the incomplete plans are needed with local search algorithms only. The systematic search algorithms can be implemented to keep track of the alternative extensions of an incomplete plan, and therefore the backup steps are not needed. Further, for these algorithms it suffices to keep track of the results of the state obtained by progression or the formula obtained by regression.

The states generated by progression from the initial state, and the formulae generated by regression are not the only possibilities for defining the search space for a search algorithm. In partial-order planning [McAllester and Rosenblitt, 1991], the search space consists of incomplete plans that are partially ordered multisets of operators. The neighbors of an incomplete plan are those obtained by adding or removing an operator, or by adding or removing an ordering constraint. Another form of incomplete plans is fixed length sequences of operators, with zero or more of the operators missing. This has been formalized as planning with propositional satisfiability as discussed in Section 3.5.

3.4 Distance estimation

Using progression and regression with just any search algorithm does not yield efficient planning. Critical for the usefulness of the algorithms is the selection of operators for the progression and regression steps. If the operators are selected randomly it is unlikely that search in possibly huge transition graphs is going to end quickly.

Operator selection can be substantially improved by using estimates on the distance between the initial state and the current goal states, for backward search, or the distance between the current state and the set of goal states, for forward search. Computing exact distances is computationally just as difficult as solving the planning problem itself. Therefore in order to speed up planning by distance information, its computation should be inexpensive, and this means that only inaccurate estimates of the distances can be used.

We present a method for distance estimation that generalizes the work of Bonet and Geffner [2001] to operators with conditional effects and arbitrary propositional formulae as preconditions.

The set makestrue(l, O), consisting of formulae ϕ such that if ϕ is true then applying an operator $o \in O$ can make the literal l true, is defined on the basis of $EPC_l(o)$ from Definition 3.3.

$$\text{makestrue}(l, O) = \{EPC_l(o) | o \in O\}$$

Example 3.15 Let $\langle A \wedge B, R \wedge (Q \rhd P) \wedge (R \rhd P) \rangle$ be an operator in O. Then $A \wedge B \wedge (Q \vee R) \in$ makestrue(P, O) because for P to become true it suffices that the precondition $A \wedge B$ of the operator and one of the antecedents Q or R of a conditional effect is true. ∎

The idea of the method for estimating distances of goal states is based on the estimation of distances of states in which given state variables have given values. The estimates are not accurate for two reasons. First, and more importantly, distance estimation is done one state variable at a time and dependencies between values of different state variables are ignored. Second, tests whether a formula is true in a set of states described by a set of literals is performed by an algorithm that approximates NP-hard satisfiability testing. Of course, because we are interested in computing distance estimates efficiently, that is in polynomial time, the inaccuracy is an acceptable compromise.

We give a recursive procedure that computes a lower bound on the number of operator applications that are needed for reaching from a state s a state in which given state variables $a \in A$ have a certain value. This is by computing a sequence of sets D_i of literals. The set D_i is a set of such literals that must be true in any state that has a distance $\leq i$ from the state s. If a literal l is in D_0, then l is true in s. If $l \in D_i \backslash D_{i+1}$, then l is true in all states with distance $\leq i$ and l may be false in some states having distance $> i$.

Definition 3.16 *Let $L = A \cup \{\neg a | a \in A\}$ be the set of literals on A. Define the sets D_i for $i \geq 0$ as follows.*

$$
\begin{aligned}
D_0 &= \{l \in L | s \models l\} \\
D_i &= D_{i-1} \backslash \{l \in L | o \in O, canbetruein(EPC_{\bar{l}}(o), D_{i-1})\}
\end{aligned}
$$

Because we consider only finite sets A of state variables and $|D_0| = |A|$ and $D_{i+1} \subseteq D_i$ for all $i \geq 0$, necessarily $D_i = D_{i+1}$ for some $i \leq |A|$.

Above canbetruein(ϕ, D) is a function that tests whether there is a state in which ϕ and the literals D are true, that is, whether $\{\phi\} \cup D$ is satisfiable. This algorithm does not accurately test satisfiability, and may claim tht $\{\phi\} \cup D$ is satisfiable even when it is not. Hence it only approximates the NP-complete satisfiability problem. The algorithm runs in polynomial time and is defined as follows.

$$
\begin{aligned}
canbetruein(\bot, D) &= false \\
canbetruein(\top, D) &= true \\
canbetruein(a, D) &= true \text{ iff } \neg a \notin D \text{ (for state variables } a \in A) \\
canbetruein(\neg a, D) &= true \text{ iff } a \notin D \text{ (for state variables } a \in A) \\
canbetruein(\neg\neg\phi, D) &= canbetruein(\phi, D) \\
canbetruein(\phi \vee \psi, D) &= canbetruein(\phi, D) \text{ or } canbetruein(\psi, D) \\
canbetruein(\phi \wedge \psi, D) &= canbetruein(\phi, D) \text{ and } canbetruein(\psi, D) \\
canbetruein(\neg(\phi \vee \psi), D) &= canbetruein(\neg\phi, D) \text{ and } canbetruein(\neg\psi, D) \\
canbetruein(\neg(\phi \wedge \psi), D) &= canbetruein(\neg\phi, D) \text{ or } canbetruein(\neg\psi, D)
\end{aligned}
$$

The reason why the satisfiability test is not accurate is that for formulae $\phi \wedge \psi$ (respectively $\neg(\phi \vee \psi)$) we make recursively two satisfiability tests that do not assume that the component formulae ϕ and ψ (respectively $\neg\phi$ and $\neg\psi$) are *simultaneously* satisfiable.

We give a lemma that states the connection between canbetruein(ϕ, D) and the satisfiabilty of $\{\phi\} \cup D$.

Lemma 3.17 *Let ϕ be a formula and D a consistent set of literals (it contains at most one of a and $\neg a$ for every $a \in A$.) If $D \cup \{\phi\}$ is satisfiable, then canbetruein(ϕ, D) returns true.*

Proof: The proof is by induction on the structure of ϕ.

Base case 1, $\phi = \bot$: The set $D \cup \{\bot\}$ is not satisfiable, and hence the implication trivially holds.

Base case 2, $\phi = \top$: canbetruein$(\top, D)$ always returns true, and hence the implication trivially holds.

Base case 3, $\phi = a$ for some $a \in A$: If $D \cup \{a\}$ is satisfiable, then $\neg a \notin D$, and hence canbetruein(a, D) returns true.

Base case 4, $\phi = \neg a$ for some $a \in A$: If $D \cup \{\neg a\}$ is satisfiable, then $a \notin D$, and hence canbetruein$(\neg a, D)$ returns true.

Inductive case 1, $\phi = \neg\neg\phi'$ for some ϕ': The formulae are logically equivalent, and by the induction hypothesis we directly establish the claim.

Inductive case 2, $\phi = \phi' \vee \psi'$: If $D \cup \{\phi' \vee \psi'\}$ is satisfiable, then either $D \cup \{\phi'\}$ or $D \cup \{\psi'\}$ is satisfiable and by the induction hypothesis at least one of canbetruein(ϕ', D) and canbetruein(ψ', D) returns true. Hence canbetruein$(\phi' \vee \psi', D)$ returns true.

Inductive case 3, $\phi = \phi' \wedge \psi'$: If $D \cup \{\phi' \wedge \psi'\}$ is satisfiable, then both $D \cup \{\phi'\}$ and $D \cup \{\psi'\}$ are satisfiable and by the induction hypothesis both canbetruein(ϕ', D) and canbetruein(ψ', D) return true. Hence canbetruein$(\phi' \wedge \psi', D)$ returns true.

Inductive cases 4 and 5, $\phi = \neg(\phi' \vee \psi')$ and $\phi = \neg(\phi' \wedge \psi')$: Like cases 2 and 3 by logical equivalence. $\square$

The other direction of the implication does not hold because for example canbetruein$(A \wedge \neg A, D)$ returns true even though the formula is not satisfiable. The procedure is a polynomial-time approximation of the logical consequence test from a set of literals: canbetruein(ϕ, D) always returns true if $D \cup \{\phi\}$ is satisfiable, but it may return true also when the set is not satisfiable.

Now we define the distances of states in which a literal l is true by $\delta_s(l) = 0$ if and only if $l \in D_0$, and for $d \geq 1$, $\delta_s(l) = d$ if and only if $\bar{l} \in D_{d-1} \backslash D_d$. For formulae ϕ we similarly define $\delta_s(\phi) = 0$ if canbetruein(ϕ, D_0), and for $d \geq 1$, $\delta_s(\phi) = d$ if canbetruein(ϕ, D_d) and not canbetruein(ϕ, D_{d-1}).

Lemma 3.18 *Let s be a state and $D_0, D_1, \ldots$ the sets given in Definition 3.16 for s. If s' is the state reached from s by applying the operator sequence $o_1, \ldots, o_n$, then $s' \models D_n$.*

Proof: By induction on n.

Base case $n = 0$: The length of the operator sequence is zero, and hence $s' = s$. The set D_0 consists exactly of those literals that are true in s, and hence $s' \models D_0$.

Inductive case $n \geq 1$: Let s'' be the state reached from s by applying $o_1, \ldots, o_{n-1}$. Now $s' = app_{o_n}(s'')$. By the induction hypothesis $s'' \models D_{n-1}$.

Let l be any literal in D_n. We show it is true in s'. Because $l \in D_n$ and $D_n \subseteq D_{n-1}$, also $l \in D_{n-1}$, and hence by the induction hypothesis $s'' \models l$.

Let ϕ be any member of makestrue$(\bar{l}, \{o_n\})$. Because $l \in D_n$ it must be that canbetruein(ϕ, D_{n-1}) returns false (Definition of D_n). Hence $D_{n-1} \cup \{\phi\}$ is by Lemma 3.17 not satisfiable, and $s'' \not\models \phi$. Hence applying o_n in s'' does not make l false, and consequently $s' \models l$.

$\square$

Theorem 3.19 *Let s be a state, ϕ a formula, and $D_0, D_1, \ldots$ the sets given in Definition 3.16 for s. If s' is the state reached from s by applying the operators $o_1, \ldots, o_n$ and $s' \models \phi$ for any formula ϕ, then canbetruein(ϕ, D_n) returns true.*

Proof: By Lemma 3.18 $s' \models D_n$. By assumption $s' \models \phi$. Hence $D_n \cup \{\phi\}$ is satisfiable. By Lemma 3.17 canbetruein(ϕ, D_n) returns true. $\square$

Corollary 3.20 *Let s be a state and ϕ a formula. Then for any sequence $o_1, \ldots, o_n$ of operators such that executing them in s results in state s' such that $s' \models \phi$, $n \geq \delta_s(\phi)$.*

Example 3.21 Consider the blocks world with three blocks and the initial state in which A is on B and B is on C.

$$D_0 \quad = \quad \{\text{A-CLEAR, A-ON-B, B-ON-C, C-ON-TABLE}, \neg\text{A-ON-C}, \neg\text{B-ON-A}, \neg\text{C-ON-A},$$
$$\neg\text{C-ON-B}, \neg\text{A-ON-TABLE}, \neg\text{B-ON-TABLE}, \neg\text{B-CLEAR}, \neg\text{C-CLEAR}\}$$

There is only one operator applicable, that moves A onto the table. Applying this operator makes the literals B-CLEAR and A-ON-TABLE and $\neg$A-ON-B true, and consequently their complementary literals do not occur in D_1, because it is possible after at most 1 operator application that these complementary literals are false.

$$D_1 \quad = \quad \{\text{A-CLEAR, B-ON-C, C-ON-TABLE}, \neg\text{A-ON-C}, \neg\text{B-ON-A}, \neg\text{C-ON-A},$$
$$\neg\text{C-ON-B}, \neg\text{B-ON-TABLE}, \neg\text{C-CLEAR}\}$$

In addition the operator applicable in the initial states, now there are three more operators applicable (their precondition does not contradict D_1), one moving A from the table on top of B (returning to the initial state), one moving B from the top of C onto A, and one moving B from the top of C onto the table. Hence D_2 is as follows.

$$D_2 \quad = \quad \{\text{C-ON-TABLE}, \neg\text{A-ON-C}, \neg\text{C-ON-A}, \neg\text{C-ON-B}\}$$

Now there are three further operators applicable, those moving C from the table onto A and onto B, and the operator moving A onto C. Consequently,

$$D_3 \quad = \quad \emptyset$$

$\blacksquare$

The next two examples demonstrate the best-case and worst-case scenarios for distance estimation.
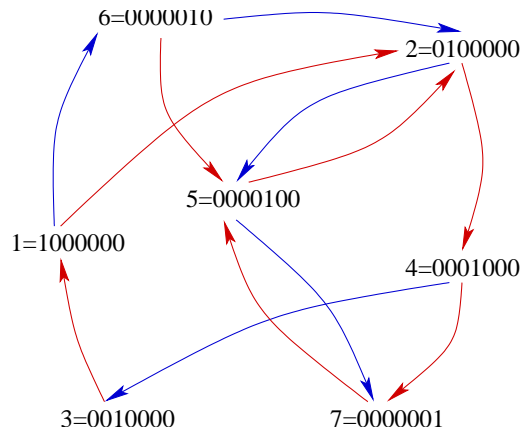
Figure 3.1: A transition system on which distance estimates are very accurate
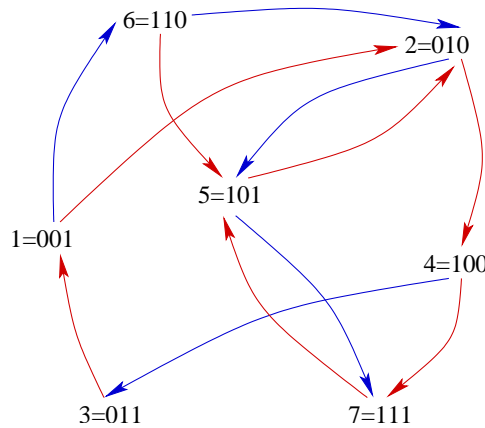

Figure 3.2: A transition system for which distance estimates are very inaccurate

Example 3.22 Figure 3.1 shows a transition system on which the distance estimates from state 1 are very accurate. The accuracy of the estimates is caused by the fact that for each state one can determine the distance accurately just on the basis of one of the state variables. Let the state variables be A, B, C, D, E, F, G.

$$
\begin{aligned}
D_0 &= \{A, \neg B, \neg C, \neg D, \neg E, \neg F, \neg G\} \\
D_1 &= \{\neg C, \neg D, \neg E, \neg G\} \\
D_2 &= \{\neg C, \neg G\} \\
D_3 &= \emptyset \\
D_4 &= \emptyset
\end{aligned}
$$

∎

Example 3.23 Figure 3.2 shows a transition system on which the distance estimates from state 1 are very poor. The inaccuracy is caused by the fact that all possible values of state variables are possible after taking just one action, and this immediately gives distance estimate 1 for all states

and sets of states.

$$
\begin{aligned}
D_0 &= \{\neg A, \neg B, C\} \\
D_1 &= \emptyset \\
D_2 &= \emptyset
\end{aligned}
$$

∎

The way Bonet and Geffner [2001] express the method differs from our presentation. Their definition is based on two mutually recursive equations that cannot be directly understood as executable procedures. The basic equation for distance computation for literals (negated and unnegated) state variables l ($l = a$ or $l = \neg a$ for some $a \in A$) is as follows (the generalization of this and the following definitions to arbitrary preconditions and conditional effects are due to us.)

$$
\delta_s(l) = \begin{cases} 0 & \text{if } s \models l \\ \min_{\phi \in \text{makestrue}(l,O)} (1 + \delta_s(\phi)) & \text{otherwise} \end{cases}
$$

The equation gives a cost/distance estimate of making a proposition $a \in A$ true starting from state s, in terms of s and cost $\delta_s(\phi)$ of reaching a state that satisfies ϕ. This cost $\delta_s(\phi)$ is defined as follows in terms of the costs $\delta_s(a)$.

$$
\begin{aligned}
\delta_s(\bot) &= \infty \\
\delta_s(\top) &= 0 \\
\delta_s(a) &= \delta_s(a) \text{ for state variables } a \in A \\
\delta_s(\neg a) &= \delta_s(\neg a) \text{ for state variables } a \in A \\
\delta_s(\neg\neg\phi) &= \delta_s(\phi) \\
\delta_s(\phi \vee \psi) &= \min(\delta_s(\phi), \delta_s(\psi)) \\
\delta_s(\phi \wedge \psi) &= \max(\delta_s(\phi), \delta_s(\psi)) \\
\delta_s(\neg(\phi \vee \psi)) &= \delta_s(\neg\phi \wedge \neg\psi) \\
\delta_s(\neg(\phi \wedge \psi)) &= \delta_s(\neg\phi \vee \neg\psi)
\end{aligned}
$$

This representation of the estimation method is useful because Bonet and Geffner [2001] have also considered another way of defining the cost of achieving a conjunction $\phi \wedge \psi$. Instead of taking the maximum of the costs of ϕ and ψ, Bonet and Geffner suggest taking the sum of the costs, which is simply obtained by replacing $\max(\delta_s(\phi), \delta_s(\psi))$ in the above equations by $\delta_s(\phi) + \delta_s(\psi)$. They call this the *additive heuristic*, in contrast to the definition given above for the *max heuristic*. The justification for this is that the max heuristic assumes that it is the cost of the more difficult conjunct that alone determines the difficulty of reaching the conjunction and the cost of the less difficult conjuncts are ignored completely. The experiments Bonet and Geffner conducted showed that the additive heuristic may lead to much more efficient planning. However, one should notice that the additive heuristic is not admissible, and indeed, Bonet and Geffner have used the max heuristic and the additive heuristic in connection with the non-optimal best-first search algorithm.

3.5 Planning as satisfiability in the propositional logic

A very powerful approach to deterministic planning emerged starting in 1992 from the work by Kautz and Selman [1992; 1996]: translate problem instances to propositional formulae $\phi_0, \phi_1, \phi_2, \ldots$ so that every valuation that satisfies formula ϕ_i corresponds to a plan of length i. Now an algorithm for testing the satisfiability of propositional formulae can be used for finding a plan: test the

satisfiability of ϕ_0, if it is unsatisfiable, continue with ϕ_1, ϕ_2, and so on, until a satisfiable formula ϕ_n is found. From the valuation the satisfiability algorithm returns we can now construct a plan of length n.

3.5.1 Actions as propositional formulae

First we need to represent all our actions in the propositional logic. We can view arbitrary propositional formulae as actions, or we can translate operators into formulae in the propositional logic. We discuss both of these possibilities.

Given a set of state variables $A = \{a_1, \ldots, a_n\}$, one could describe an action directly as a propositional formula ϕ over propositions $A \cup A'$ where $A' = \{a'_1, \ldots, a'_n\}$. Here the propositions A represent the values of state variables in the state s in which an action is taken, and propositions A' the values of state variables in a successor state s'.

A pair of valuations s and s' can be understood as a valuation of $A \cup A'$ (the state s assigns a value to propositions A and s' to propositions A'), and a transition from s to s' is possible if and only if $s, s' \models \phi$.

Example 3.24 Let there be state variables a_1 and a_2. The action that reverses the values of both state variables is described by $(a_1 \leftrightarrow \neg a'_1) \wedge (a_2 \leftrightarrow \neg a'_2)$.

This action is represented by the following matrix.

	$a'_1 a'_2$ $=$ $0\,0$	$a'_1 a'_2$ $=$ $0\,1$	$a'_1 a'_2$ $=$ $1\,0$	$a'_1 a'_2$ $=$ $1\,1$
$a_1 a_2 = 00$	0	0	0	1
$a_1 a_2 = 01$	0	0	1	0
$a_1 a_2 = 10$	0	1	0	0
$a_1 a_2 = 11$	1	0	0	0

The matrix can be equivalently represented as the following truth-table.

$a_1 a_2 a'_1 a'_2$	
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
0 1 0 0	0
0 1 0 1	0
0 1 1 0	1
0 1 1 1	0
1 0 0 0	0
1 0 0 1	1
1 0 1 0	0
1 0 1 1	0
1 1 0 0	1
1 1 0 1	0
1 1 1 0	0
1 1 1 1	0

Of course, this is the truth-table of $(a_1 \leftrightarrow \neg a'_1) \wedge (a_2 \leftrightarrow \neg a'_2)$.   ∎

Example 3.25 Let the set of state variables be $A = \{a_1, a_2, a_3\}$. The formula $(a_1 \leftrightarrow a'_2) \wedge (a_2 \leftrightarrow a'_3) \wedge (a_3 \leftrightarrow a'_1)$ represents the action that rotates the values of the state variables a_1, a_2 and a_3 one position right. The formula can be represented as the following adjacency matrix. The rows correspond to valuations of A and the columns to valuations of $A' = \{a'_1, a'_2, a'_3\}$.

	000	001	010	011	100	101	110	111
000	1	0	0	0	0	0	0	0
001	0	0	0	0	1	0	0	0
010	0	1	0	0	0	0	0	0
011	0	0	0	0	0	1	0	0
100	0	0	1	0	0	0	0	0
101	0	0	0	0	0	0	1	0
110	0	0	0	1	0	0	0	0
111	0	0	0	0	0	0	0	1

A more conventional way of depicting the valuations of this formula would be as a truth-table with one row for every valuation of $A \cup A'$, a total of 64 rows.   ∎

This kind of propositional formulae are the basis of a number of planning algorithms that are based on reasoning in propositional logics. These formulae could be input to a planning algorithm, but describing actions in that way is usually more tricky than as operators, and these formulae are usually just automatically derived from operators.

The action in Example 3.25 is deterministic. Not all actions represented by propositional formulae are deterministic. A sufficient (but not necessary) condition for the determinism is that the formula is of the form $(\phi_1 \leftrightarrow a'_1) \wedge \cdots \wedge (\phi_n \leftrightarrow a'_n)$ with exactly one equivalence for every $a' \in A'$ and formulae ϕ_i not having occurrences of propositions in A'. This way the truth-value of every state variable in the successor state is unambiguously defined in terms of the truth-values of the state variables in the predecessor state, and hence the operator is deterministic.

3.5.2 Translation of operators into propositional logic

We first give the simplest possible translation of deterministic planning into the propositional logic. In this translation every operator is separately translated into a formula, and the choice between the operators can be represented by the disjunction connective of the propositional logic.

The formula τ_o that represents operator $o = \langle z, e \rangle$ is the conjunction of the precondition z and the formulae

$$((EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))) \leftrightarrow a') \wedge \neg(EPC_a(e) \wedge EPC_{\neg a}(e))$$

for every $a \in A$. Above the first conjunct expresses the value of a in the successor state in terms of the values of the state variables in the predecessor state. This is like in the definition of regression in Section 3.2.2. The second conjunct says that applying the operator is not possible if it assigns both the value 1 and 0 to a.

Example 3.26 Consider operator $\langle A \vee B, ((B \vee C) \rhd A) \wedge (\neg C \rhd \neg A) \wedge (A \rhd B) \rangle$.

The corresponding propositional formula is

$$(A \vee B) \quad \wedge(((B \vee C) \vee (A \wedge \neg\neg C)) \leftrightarrow A') \wedge \neg((B \vee C) \wedge \neg C)$$
$$\wedge((A \vee (B \wedge \neg\bot)) \leftrightarrow B') \wedge \neg(A \wedge \bot)$$
$$\wedge((\bot \vee (C \wedge \neg\bot)) \leftrightarrow C') \wedge \neg(\bot \wedge \bot)$$
$$\equiv$$
$$(A \vee B) \quad \wedge(((B \vee C) \vee (A \wedge C)) \leftrightarrow A') \wedge \neg((B \vee C) \wedge \neg C)$$
$$\wedge((A \vee B) \leftrightarrow B')$$
$$\wedge(C \leftrightarrow C')$$

∎

Applying any of the operators $o_1, \ldots, o_n$ or none of the operators is now represented as the formula

$$\mathcal{R}_1(A, A') = \tau_{o_1} \vee \cdots \vee \tau_{o_n} \vee ((a_1 \leftrightarrow a'_1) \wedge \cdots \wedge (a_k \leftrightarrow a'_k))$$

where $A = \{a_1, \ldots, a_k\}$ is the set of all state variables. The last disjunct is for the case that no operator is applied.

The valuations that satisfy this formula do not uniquely determine which operator was applied, because for a given state two operators may produce the same successor state. However, in such cases it usually does not matter which operator is applied and one of them can be chosen arbitrarily.

3.5.3 Finding plans by satisfiability algorithms

We show how plans can be found by first translating problem instances $\langle A, I, O, G \rangle$ into propositional formulae, and then finding satisfying valuations by a satisfiability algorithm.

In Section 3.5.1 we showed how operators can be described by propositional formulae over sets A and A' of propositions, the set A describing the values of the state variables in the state in which the operator is applied, and the set A' describing the values of the state variables in the successor state of that state.

Now, for a fixed plan length n, we define sets of propositions $A_0, \ldots, A_n$ with propositions in A_i describing the values of the state variables at time point i, that is, when i operators (or sets of operators, if we have parallelism) have been applied.

Let $\langle A, I, O, G \rangle$ be a problem instance in deterministic planning.

The state at the first time point 0 is determined by I, and at the last time point n a goal state must have been reached. Therefore we include ι with time-labeling 0 and G with time-labeling n in the encoding.

$$\iota^0 \wedge \mathcal{R}_1(A_0, A_1) \wedge \mathcal{R}_1(A_1, A_2) \wedge \cdots \wedge \mathcal{R}_1(A_{n-1}, A_n) \wedge G^n$$

Here $\iota^0 = \bigwedge\{a^0 | a \in A, I(a) = 1\} \cup \{\neg a^0 | a \in A, I(a) = 0\}$ and G^n is G with propositions a replaced by a^n.

Plans are found incrementally by increasing the plan length and testing the satisfiability of the corresponding formulae: first try to find plans of length 0, then of length 1, 2, 3, and so on, until a plan is found. If there are no plans, it has to be somehow decided when to stop increasing the plan length that is tried. An upper bound on plan length is $2^{|A|} - 1$ where A is the set of state variables, but this upper bound does not provide a practical termination condition for this procedure.

The size of the encoding is linear in the plan length, and because the plan length may be exponential, the encoding might not be practical for very long plans, as runtimes of satisfiability algorithms in general grow exponentially in the length of the formulae.

Example 3.27 Consider an initial state that satisfies $I \models A \wedge B$, the goal $G = (A \wedge \neg B) \vee (\neg A \wedge B)$, and the operators $o_1 = \langle \top, (A \rhd \neg A) \wedge (\neg A \rhd A) \rangle$ and $o_2 = \langle \top, (B \rhd \neg B) \wedge (\neg B \rhd B) \rangle$.

The following formula is satisfiable if and only if $\langle A, I, \{o_1, o_2\}, G \rangle$ has a plan of length 3 or less.

$(A^0 \wedge B^0)$
$\wedge (((A^0 \leftrightarrow A^1) \wedge (B^0 \leftrightarrow \neg B^1)) \vee ((A^0 \leftrightarrow \neg A^1) \wedge (B^0 \leftrightarrow B^1)) \vee ((A^0 \leftrightarrow A^1) \wedge (B^0 \leftrightarrow B^1)))$
$\wedge (((A^1 \leftrightarrow A^2) \wedge (B^1 \leftrightarrow \neg B^2)) \vee ((A^1 \leftrightarrow \neg A^2) \wedge (B^1 \leftrightarrow B^2)) \vee ((A^1 \leftrightarrow A^2) \wedge (B^1 \leftrightarrow B^2)))$
$\wedge (((A^2 \leftrightarrow A^3) \wedge (B^2 \leftrightarrow \neg B^3)) \vee ((A^2 \leftrightarrow \neg A^3) \wedge (B^2 \leftrightarrow B^3)) \vee ((A^2 \leftrightarrow A^3) \wedge (B^2 \leftrightarrow B^3)))$
$\wedge ((A^3 \wedge \neg B^3) \vee (\neg A^3 \wedge B^3))$

One of the valuations that satisfy the formula is the following.

	time i			
	0	1	2	3
A^i	1	0	0	0
B^i	1	1	0	1

This valuation corresponds to the plan that applies operator o_1 at time point 0, o_2 at time point 1, and o_2 at time point 2. There are also other satisfying valuations. The shortest plans for this problem instance are o_1 and o_2, each consisting of one operator only. ∎

Example 3.28 Consider the following problem. There are two operators, one for rotating the values of bits ABC one step right, and the other for inverting the values of all the bits. Consider reaching from the initial state 100 the goal state 001 with two actions. This is represented as the following formula.

$(A^0 \wedge \neg B^0 \wedge \neg C^0)$
$\wedge (((A^0 \leftrightarrow B^1) \wedge (B^0 \leftrightarrow C^1) \wedge (C^0 \leftrightarrow A^1)) \vee ((\neg A^0 \leftrightarrow A_1) \wedge (\neg B^0 \leftrightarrow B^1) \wedge (\neg C^0 \leftrightarrow C^1)))$
$\wedge (((A^1 \leftrightarrow B^2) \wedge (B^1 \leftrightarrow C^2) \wedge (C^1 \leftrightarrow A^2)) \vee ((\neg A^1 \leftrightarrow A^2) \wedge (\neg B^1 \leftrightarrow B^2) \wedge (\neg C^1 \leftrightarrow C^2)))$
$\wedge (\neg A^2 \wedge \neg B^2 \wedge C^2)$

Because the literals describing the initial and the goal state must be true, we can replace other occurrences of these state variables by $\top$ and $\bot$.

$(A^0 \wedge \neg B^0 \wedge \neg C^0)$
$\wedge (((\top \leftrightarrow B^1) \wedge (\bot \leftrightarrow C^1) \wedge (\bot \leftrightarrow A^1)) \vee ((\neg \top \leftrightarrow A_1) \wedge (\neg \bot \leftrightarrow B^1) \wedge (\neg \bot \leftrightarrow C^1)))$
$\wedge (((A^1 \leftrightarrow \bot) \wedge (B^1 \leftrightarrow \top) \wedge (C^1 \leftrightarrow \bot)) \vee ((\neg A^1 \leftrightarrow \bot) \wedge (\neg B^1 \leftrightarrow \bot) \wedge (\neg C^1 \leftrightarrow \top)))$
$\wedge (\neg A^2 \wedge \neg B^2 \wedge C^2)$

After simplifying we have the following.

$(A^0 \wedge \neg B^0 \wedge \neg C^0)$
$\wedge ((B^1 \wedge \neg C^1 \wedge \neg A^1) \vee (\neg A_1 \wedge B^1 \wedge C^1)$
$\wedge ((\neg A^1 \wedge B^1 \wedge \neg C^1) \vee (A^1 \wedge B^1 \wedge \neg C^1))$
$\wedge (\neg A^2 \wedge \neg B^2 \wedge C^2)$

Clearly, the only way of satisfying this formula is to make the first disjuncts of both disjunctions true, that is, B^1 must be true and A^1 and C^1 must be false.

The resulting valuation corresponds to taking the rotation action twice.
Consider the same problem but now with the goal state 101.

$(A^0 \wedge \neg B^0 \wedge \neg C^0)$
$\wedge(((A^0 \leftrightarrow B^1) \wedge (B^0 \leftrightarrow C^1) \wedge (C^0 \leftrightarrow A^1)) \vee ((\neg A^0 \leftrightarrow A_1) \wedge (\neg B^0 \leftrightarrow B^1) \wedge (\neg C^0 \leftrightarrow C^1)))$
$\wedge(((A^1 \leftrightarrow B^2) \wedge (B^1 \leftrightarrow C^2) \wedge (C^1 \leftrightarrow A^2)) \vee ((\neg A^1 \leftrightarrow A^2) \wedge (\neg B^1 \leftrightarrow B^2) \wedge (\neg C^1 \leftrightarrow C^2)))$
$\wedge(A^2 \wedge \neg B^2 \wedge C^2)$

We simplify again and get the following formula.

$$(A^0 \wedge \neg B^0 \wedge \neg C^0)$$
$$\wedge((B^1 \wedge \neg C^1 \wedge \neg A^1) \vee (\neg A_1 \wedge B^1 \wedge C^1))$$
$$\wedge((\neg A^1 \wedge B^1 \wedge C^1) \vee (\neg A^1 \wedge B^1 \wedge \neg C^1))$$
$$\wedge(A^2 \wedge \neg B^2 \wedge C^2)$$

Now there are two possible plans, to rotate first and then invert the values, or first invert and then rotate. These respectively correspond to making the first disjunct of the first disjunction and the second disjunct of the second disjunction true, or the second and the first disjunct. ∎

3.5.4 Parallel plans

Plans so far always have had one operator at a time point. It turns out that it is often useful to allow *several operators in parallel*. This is beneficial for two main reasons.

First, consider a number of operators that affect and depend on disjoint state variables so that they can be applied in any order. If there are n such operators, there are $n!$ plans that are equivalent in the sense that each leads to the same state. When a satisfiability algorithm is used in showing that there is no plan of length n consisting of these operators, it has to show that none of the $n!$ plans reaches the goals. This may be combinatorially very difficult if n is high.

Second, when several operators can be applied simultaneously, it is not necessary to represent all intermediate states of the corresponding sequential plans: parallel plans require less time points than the corresponding sequential plans.

For sequences $o_1; o_2; \ldots; o_n$ of operators we define $app_{o_1;o_2;\ldots;o_n}(s)$ as $app_{o_n}(\cdots app_{o_2}(app_{o_1}(s))\cdots)$. For sets S of operators and states s we define $app_S(s)$ as the result of simultaneously applying all operators $o \in S$: the preconditions of all operators in S must be true in s and the state $app_S(s)$ is obtained from s by making the literals in $\bigcup_{\langle p,e\rangle \in S}([e]_s)$ true. Analogously to sequential plans we can define $app_{S_1;S_2;\ldots;S_n}(s)$ as $app_{S_n}(\cdots app_{S_2}(app_{S_1}(s))\cdots)$.

Definition 3.29 (Step plans) *For a set of operators O and an initial state I, a plan is a sequence $T = S_1, \ldots, S_l$ of sets of operators such that there is a sequence of states $s_0, \ldots, s_l$ (the execution of T) such that*

1. $s_0 = I$,

2. $\bigcup_{\langle p,e\rangle \in S_i}([e]_{s_{i-1}})$ *is consistent for every* $i \in \{1, \ldots, l\}$,

3. $s_i = app_{S_i}(s_{i-1})$ *for* $i \in \{1, \ldots, l\}$,

4. *for all* $i \in \{1, \ldots, l\}$ *and* $\langle p, e\rangle = o \in S_i$ *and* $S \subseteq S_i\backslash\{o\}$, $app_S(s_{i-1}) \models p$, *and*

5. *for all* $i \in \{1, \ldots, l\}$ *and* $\langle p, e\rangle = o \in S_i$ *and* $S \subseteq S_i\backslash\{o\}$, $[e]_{s_{i-1}} = [e]_{app_S(s_{i-1})}$.

The last condition says that the changes an operator makes would be the same also if some of the operators parallel to it would have been applied before it. This means that the parallel application can be understood as applying the operators in any order, with the requirement that the state that is reached is the same in every case.

Indeed, we can show that a parallel plan can be linearized in an arbitrary way, without affecting which state it reaches.

Lemma 3.30 *Let $T = S_1, \ldots, S_k, \ldots, S_l$ be a step plan. Let $T' = S_1, \ldots, S_k^0, S_k^1, \ldots, S_l$ be the step plan obtained from T by splitting the step S_k into two steps S_k^0 and S_k^1 such that $S_k = S_k^0 \cup S_k^1$ and $S_k^0 \cap S_k^1 = \emptyset$.*

If $s_0, \ldots, s_k, \ldots, s_l$ is the execution of T then $s_0, \ldots, s_k', s_k, \ldots, s_l$ for some s_k' is the execution of T'.

Proof: So $s_k' = app_{S_k^0}(s_{k-1})$ and $s_k = app_{S_k}(s_{k-1})$ and we have to prove that $app_{S_k^1}(s_k') = s_k$. We will show that the active effects of every operator $o \in S_k^1$ are the same in s_{k-1} and in s_k', and hence the changes from s_{k-1} to s_k are the same in both plans. Let $o^1, \ldots, o^z$ be the operators in S_k^0, and let $T_i = \{o^1, \ldots, o^i\}$ for every $i \in \{0, \ldots, z\}$. We show by induction that changes caused by every operator $o \in S_k^1$ are the same when executed in s_{k-1} and in $app_{T_i}(s_{k-1})$, from which the claim follows because $s_k' = app_{T_z}(s_{k-1})$.

Base case $i = 0$: Immediate because $T_0 = \emptyset$.

Inductive case $i \geq 1$: By the induction hypothesis the changes caused by every $o \in S_k^1$ are the same when executed in s_{k-1} and in $app_{T_{i-1}}(s_{k-1})$. In $app_{T_i}(s_{k-1})$ additionally the operator o^i has been applied. We have to show that this operator application does affect the set of active effects of o. By the definition of step plans, $[e]_{app_{T_{i-1}}(s_{k-1})} = [e]_{app_{T_{i-1} \cup \{o^i\}}(s_{k-1})}$. This establishes the induction hypothesis and completes the proof. $\square$

Theorem 3.31 *Let $T = S_1, \ldots, S_k, \ldots, S_l$ be a step plan. Then any $\sigma = o_1^1; \ldots; o_{n_1}^1; o_2^2; \ldots; o_{n_2}^2; \ldots; o_1^l; \ldots; o_{n_l}^l$ such that for every $i \in \{1, \ldots, l\}$ the sequence $o_1^i; \ldots; o_{n_i}^i$ is a total ordering of S_i, is a plan, and its execution leads to the same terminal state as that of T.*

Proof: First, all empty steps can be removed from the step plan. By Lemma 3.30 non-singleton steps can be split repeatedly to two smaller non-empty steps until every step is singleton and the singleton steps are in the desired order. The resulting plan is a sequential plan. $\square$

Lemma 3.32 *Testing whether a sequence of sets of operators is a parallel plan is co-NP-hard.*

Proof: We can reduce the NP-complete satisfiability problem of the propositional logic to it. Let ϕ be a propositional formula in which the propositional variables $A = \{a_1, \ldots, a_n\}$ occur. Let I be an initial state in which all state variables are false. Now ϕ is valid if and only if $S_1 = \{\langle \top, \phi \triangleright A \rangle, \langle \top, a_1 \rangle, \langle \top, a_2 \rangle, \ldots, \langle \top, a_n \rangle\}$ is a parallel plan that reaches the goal A. $\square$

However, there are simple sufficient conditions that guarantee that a sequence of sets of operators satisfies the definition of parallel plans. A commonly used condition is that a state variable affected by any of the operators at one step of a plan does not occur in the precondition or in the antecedent of a conditional of any other operator in that step.

3.5.5 Translation of parallel planning into propositional logic

The second translation we give allows applying several operators is parallel. The translation differs from the one in Section 3.5.2 in that the translation is not obtained simply by combining the translations of individual operators, and that we use propositions for explicitly representing which operators are applied.

Let $o_1, \ldots, o_m$ be the operators, and $e_1, \ldots, e_m$ their respective effects. Let $a \in A$ be one of the state variables. Then we have the following formulae expressing the conditions under which the state variable p may change from false to true and from true to false.

$$(\neg a \wedge a') \rightarrow ((o_1 \wedge EPC_a(e_1)) \vee \cdots \vee (o_m \wedge EPC_a(e_m)))$$
$$(a \wedge \neg a') \rightarrow ((o_1 \wedge EPC_{\neg a}(e_1)) \vee \cdots \vee (o_m \wedge EPC_{\neg a}(e_m)))$$

Further, for every operator $\langle z, e \rangle \in O$ we have formulae that describe what values the state variables have in the predecessor and in the successor states if the operator is applied. Then the state variables $a_1, \ldots, a_n$ may be affected as follows, and the precondition z of the operator must be true in the predecessor state.

$$\begin{aligned}
(o \wedge EPC_{a_1}(e)) &\rightarrow a'_1 \\
(o \wedge EPC_{\neg a_1}(e)) &\rightarrow \neg a'_1 \\
&\vdots \\
(o \wedge EPC_{a_n}(e)) &\rightarrow a'_n \\
(o \wedge EPC_{\neg a_n}(e)) &\rightarrow \neg a'_n \\
o &\rightarrow z
\end{aligned}$$

Example 3.33 Consider the operators $o_1 = \langle \neg LAMP1, LAMP1 \rangle$ and $o_2 = \langle \neg LAMP2, LAMP2 \rangle$. The application of none, one or both of these operators is described by the following formula.

$$\begin{aligned}
(\neg LAMP1 \wedge LAMP1') &\rightarrow ((o_1 \wedge \top) \vee (o_2 \wedge \bot) \\
(LAMP1 \wedge \neg LAMP1') &\rightarrow ((o_1 \wedge \bot) \vee (o_2 \wedge \bot) \\
(\neg LAMP2 \wedge LAMP2') &\rightarrow ((o_1 \wedge \bot) \vee (o_2 \wedge \top) \\
(LAMP2 \wedge \neg LAMP2') &\rightarrow ((o_1 \wedge \bot) \vee (o_2 \wedge \bot) \\
o_1 &\rightarrow LAMP1' \\
o_1 &\rightarrow \neg LAMP1 \\
o_2 &\rightarrow LAMP2' \\
o_2 &\rightarrow \neg LAMP2
\end{aligned}$$

∎

Finally, we have to guarantee that the last two conditions of parallel plans, that the simultaneous execution leads to the same result as executing them in any order, are satisfied. Encoding the conditions exactly is difficult, but we can use a simple encoding that provides a sufficient condition that the conditions are satisfied. We just have

$$\neg o_i \vee \neg o_j$$

whenever there is a state variable p occurring as an effect in o_i and in the precondition or the antecedent of a conditional effect of o_j.

We use

$$\mathcal{R}_2(A, A')$$

to denote the conjunction of all the above formulae.

Like $\mathcal{R}_1(A, A')$, later we use also $\mathcal{R}_2(A, A')$ with propositions labeled for different time points, and then we also have to label the propositions o for operators so that operator applications at different time points correspond to different propositions, for example o^0, o^1 and so on. For the labels for other propositions we use the superscript t in $\mathcal{R}_2^t(A, A')$.

3.5.6 Plan existence as evaluation of quantified Boolean formulae

For a more concise representation of the deterministic planning problem we need a slightly more expressive language than the propositional logic. Quantified Boolean formulae are exactly right for this purpose.

Consider the following QBF that represents the existence of transition sequences of length 2^n between two states.

$$\exists A \exists A'(\text{reach}_n(A, A') \land I \land G) \tag{3.1}$$

Here I and G are the formulae describing the initial and goal states respectively expressed in terms of variables from sets A and A'. Here $\text{reach}_i(A, A')$ means that a state represented in terms of variables from A' can be reached with $\leq 2^i$ steps from a state represented in terms of variables from A. It is recursively defined as follows.

$$
\begin{aligned}
\text{reach}_0(A, A') &\overset{\text{def}}{\equiv} \mathcal{R}_1(A, A') \\
\text{reach}_{i+1}(A, A') &\overset{\text{def}}{\equiv} \exists T \forall c \exists T_1 \exists T_2 (\text{reach}_i(T_1, T_2) \\
&\qquad \land (c \rightarrow (T_1 = A \land T_2 = T)) \\
&\qquad \land (\neg c \rightarrow (T_1 = T \land T_2 = A')))
\end{aligned}
$$

The sets T and A consist of propositional variables, and $A = T$ for $A = \{a_1, \ldots, a_n\}$ and $T = \{t_1, \ldots, t_n\}$ means $(a_1 \leftrightarrow t_1) \land \cdots \land (a_n \leftrightarrow t_n)$. The idea of the definition of $\text{reach}_{i+1}(A, A')$ is that the variables T describe a state halfway between A and A', and the two values for the variable c correspond to two reachability tests, one between A and T, and the other between T and A'.

This is how the PSPACE-hardness of evaluation of QBF can be proved, with $\mathcal{R}_1(A, A')$ representing the transitions of a deterministic polynomial-space Turing machine, see for example [Balcázar *et al.*, 1988].

If we eliminate all universal variables from Formula 3.1, we see that it is essentially a concise $\mathcal{O}(\log t)$ space ($t = 2^n$) representation of

$$I_0 \land \mathcal{R}_1(A_0, A_1) \land \mathcal{R}_1(A_1, A_2) \land \cdots \land \mathcal{R}_1(A_{t-1}, A_t) \land G_t \tag{3.2}$$

with only one occurrence of the transition relation.

The representation of deterministic planning as quantified Boolean formulae is more concise that the representation in the propositional logic, but it currently seems that the algorithms for testing the satisfiability solve the planning problem much more efficiently than algorithms for evaluating the values of QBF.

3.6 Invariants

Planning with both regression and propositional satisfiability suffer from the problem of states (valuations of state variables) that are not reachable from the initial state. Even when the number of state variables is high, the number of possible states of the world might be rather small, because not all valuations correspond to a possible world state. Hence for example regression may produce formulae that represent states that are not reachable from the initial state, and due to this backward search may spend a lot of time doing unfruitful work[1]. Clearly, search would be more efficient if backward search could be restricted to state that are indeed reachable from the initial state. Planning as propositional satisfiability suffers from the same problem.

It would be useful to eliminate those states from consideration that do not represent possible world states. However, determining whether a given state is reachable from the initial state is PSPACE-complete and equivalent to the plan existence problem of deterministic planning, and consequently computing exact information on the reachability of states could not be used for speeding up the basic forward and backward search algorithms: solving the subproblem would be just as complex as solving the problem itself, and would just lead to slow planning.

However, there is the possibility of using inexact, less expensive information about the reachability of states. In this section we present a polynomial time algorithm for computing inexact information about the reachability of states that has turned out very useful in speeding up planning algorithms based on backward search as well as other algorithms that use incomplete descriptions of sets of states, like plan search by using propositional logic in Section 3.5.

An *invariant* is a formula that holds in the initial state of a planning problem and that holds in every state that is reached by an action from a state in which it holds. A formula ϕ is *the strongest invariant* if for any invariant ψ, $\phi \models \psi$. The strongest invariant exactly characterizes the set of all states that are reachable from the initial state: For all states s, $s \models \phi$ if and only if s is reachable from the initial state. The strongest invariant is unique up to a logical equivalence.

Example 3.34 Consider a set of blocks that are on the table, and that can be stacked on top of each other so that every block can be on at most one block and on every block there can be at most one block.

We can formalize the actions that are possible in this setting as the following schematic operators.

$$\langle \text{ontable}(x) \wedge \text{clear}(x) \wedge \text{clear}(y), \text{on}(x, y) \wedge \neg \text{clear}(y) \wedge \neg \text{ontable}(x) \rangle$$
$$\langle \text{clear}(x) \wedge \text{on}(x, y), \text{ontable}(x) \wedge \text{clear}(y) \wedge \neg \text{on}(x, y) \rangle$$
$$\langle \text{clear}(x) \wedge \text{on}(x, y) \wedge \text{clear}(z), \text{on}(x, z) \wedge \text{clear}(y) \wedge \neg \text{clear}(z) \wedge \neg \text{on}(x, y) \rangle$$

When instantiated with three objects $X = \{A, B, C\}$ we get the following operators.

---
[1]A similar problem arises with forward search, because with progression one may reach states from which the goals cannot be reached.

$\langle \text{ontable}(A) \wedge \text{clear}(A) \wedge \text{clear}(B), \text{on}(A, B) \wedge \neg\text{clear}(B) \wedge \neg\text{ontable}(A) \rangle$
$\langle \text{ontable}(A) \wedge \text{clear}(A) \wedge \text{clear}(C), \text{on}(A, C) \wedge \neg\text{clear}(C) \wedge \neg\text{ontable}(A) \rangle$
$\langle \text{ontable}(B) \wedge \text{clear}(B) \wedge \text{clear}(A), \text{on}(B, A) \wedge \neg\text{clear}(A) \wedge \neg\text{ontable}(B) \rangle$
$\langle \text{ontable}(B) \wedge \text{clear}(B) \wedge \text{clear}(C), \text{on}(B, C) \wedge \neg\text{clear}(C) \wedge \neg\text{ontable}(B) \rangle$
$\langle \text{ontable}(C) \wedge \text{clear}(C) \wedge \text{clear}(A), \text{on}(C, A) \wedge \neg\text{clear}(A) \wedge \neg\text{ontable}(C) \rangle$
$\langle \text{ontable}(C) \wedge \text{clear}(C) \wedge \text{clear}(B), \text{on}(C, B) \wedge \neg\text{clear}(B) \wedge \neg\text{ontable}(C) \rangle$

$\langle \text{clear}(A) \wedge \text{on}(A, B), \text{ontable}(A) \wedge \text{clear}(B) \wedge \neg\text{on}(A, B) \rangle$
$\langle \text{clear}(A) \wedge \text{on}(A, C), \text{ontable}(A) \wedge \text{clear}(C) \wedge \neg\text{on}(A, C) \rangle$
$\langle \text{clear}(B) \wedge \text{on}(B, A), \text{ontable}(B) \wedge \text{clear}(A) \wedge \neg\text{on}(B, A) \rangle$
$\langle \text{clear}(B) \wedge \text{on}(B, C), \text{ontable}(B) \wedge \text{clear}(C) \wedge \neg\text{on}(B, C) \rangle$
$\langle \text{clear}(C) \wedge \text{on}(C, A), \text{ontable}(C) \wedge \text{clear}(A) \wedge \neg\text{on}(C, A) \rangle$
$\langle \text{clear}(C) \wedge \text{on}(C, B), \text{ontable}(C) \wedge \text{clear}(B) \wedge \neg\text{on}(C, B) \rangle$

$\langle \text{clear}(A) \wedge \text{on}(A, B) \wedge \text{clear}(C), \text{on}(A, C) \wedge \text{clear}(B) \wedge \neg\text{clear}(C) \wedge \neg\text{on}(A, B) \rangle$
$\langle \text{clear}(A) \wedge \text{on}(A, C) \wedge \text{clear}(B), \text{on}(A, B) \wedge \text{clear}(C) \wedge \neg\text{clear}(B) \wedge \neg\text{on}(A, C) \rangle$
$\langle \text{clear}(B) \wedge \text{on}(B, A) \wedge \text{clear}(C), \text{on}(B, C) \wedge \text{clear}(A) \wedge \neg\text{clear}(C) \wedge \neg\text{on}(B, A) \rangle$
$\langle \text{clear}(B) \wedge \text{on}(B, C) \wedge \text{clear}(A), \text{on}(B, A) \wedge \text{clear}(C) \wedge \neg\text{clear}(A) \wedge \neg\text{on}(B, C) \rangle$
$\langle \text{clear}(C) \wedge \text{on}(C, A) \wedge \text{clear}(B), \text{on}(C, B) \wedge \text{clear}(A) \wedge \neg\text{clear}(B) \wedge \neg\text{on}(C, A) \rangle$
$\langle \text{clear}(C) \wedge \text{on}(C, B) \wedge \text{clear}(A), \text{on}(C, A) \wedge \text{clear}(B) \wedge \neg\text{clear}(A) \wedge \neg\text{on}(C, B) \rangle$

Here a block being clear means that no block is on top of it.

Let all the blocks be initially on the table. Hence the initial state satisfies the formula

$\text{clear}(A) \wedge \text{clear}(B) \wedge \text{clear}(C) \wedge \text{ontable}(A) \wedge \text{ontable}(B) \wedge \text{ontable}(C) \wedge$
$\neg\text{on}(A, B) \wedge \neg\text{on}(A, C) \wedge \neg\text{on}(B, A) \wedge \neg\text{on}(B, C) \wedge \neg\text{on}(C, A) \wedge \neg\text{on}(C, B)$

that determines the truth-values of all state variables uniquely.

All the invariants in this problem instance are the following.

$$\text{clear}(A) \leftrightarrow (\neg\text{on}(B, A) \wedge \neg\text{on}(C, A))$$
$$\text{clear}(B) \leftrightarrow (\neg\text{on}(A, B) \wedge \neg\text{on}(C, B))$$
$$\text{clear}(C) \leftrightarrow (\neg\text{on}(A, C) \wedge \neg\text{on}(B, C))$$
$$\text{ontable}(A) \leftrightarrow (\neg\text{on}(A, B) \wedge \neg\text{on}(A, C))$$
$$\text{ontable}(B) \leftrightarrow (\neg\text{on}(B, A) \wedge \neg\text{on}(B, C))$$
$$\text{ontable}(C) \leftrightarrow (\neg\text{on}(C, A) \wedge \neg\text{on}(C, B))$$
$$\neg\text{on}(A, B) \vee \neg\text{on}(A, C)$$
$$\neg\text{on}(B, A) \vee \neg\text{on}(B, C)$$
$$\neg\text{on}(C, A) \vee \neg\text{on}(C, B)$$
$$\neg\text{on}(B, A) \vee \neg\text{on}(C, A)$$
$$\neg\text{on}(A, B) \vee \neg\text{on}(C, B)$$
$$\neg\text{on}(A, C) \vee \neg\text{on}(B, C)$$
$$\neg(\text{on}(A, B) \wedge \text{on}(B, C) \wedge \text{on}(C, A))$$
$$\neg(\text{on}(A, C) \wedge \text{on}(C, B) \wedge \text{on}(B, A))$$

The conjunction of these formulae describes exactly the set of states that are reachable from the initial state by the operators, and intuitively describes all the possible configurations the three blocks can be in.

We can schematically give the invariants for any set X of blocks as follows.

$\text{clear}(x) \leftrightarrow \forall y \in X \backslash \{x\}.\neg\text{on}(y, x)$
$\text{ontable}(x) \leftrightarrow \forall y \in X \backslash \{x\}.\neg\text{on}(x, y)$
$\neg\text{on}(x, y) \vee \neg\text{on}(x, z) \text{ when } y \neq z$
$\neg\text{on}(y, x) \vee \neg\text{on}(z, x) \text{ when } y \neq z$
$\neg(\text{on}(x_1, x_2) \wedge \text{on}(x_2, x_3) \wedge \cdots \wedge \text{on}(x_{n-1}, x) \wedge \text{on}(x_n, x_1)) \text{ for all } n \geq 1 \text{ and } \{x_1, \ldots, x_n\} \subseteq X$

The last schematic formula says that the *on* relation is acyclic. ■

Because testing whether a state satisfies all invariants, that is whether it is reachable from the initial state, is PSPACE-hard, the requirement that invariant computation is polynomial time leads to computing only invariants that are weaker than the strongest invariant. This kind of set of invariants only gives an upper bound (with respect to set-inclusion) on the set of reachable states.

The algorithm we present computes invariants that are disjunctions of at most n literals, for a fixed n. For representing all invariants, no finite upper bound on n may be imposed, but then also invariant computation could not be performed in polynomial time. Although the computation is polynomial time for any fixed n, the runtimes grow quickly as n is increased, and it is most useful for $n = 2$, that is, for invariants that are disjunctions of two literals.

The algorithm proceeds by first computing all n-literal clauses that are true in the initial state. Then, the algorithm removes all clauses that are not true after 1 operator application, after 2 operator applications, and so on, until the set of clauses does not change. At this point all the clauses are invariants and hold in all states that are reachable from the initial state.

3.6.1 Algorithms for computing invariants

Our algorithm for computing invariants has a similar flavor to distance estimation in Section 3.4: starting from a description of what is possible in the initial state, we inductively determine what is possible after i operator applications. In contrast to the distance estimation method, the states that are reachable after i operator applications are not characterized by sets of literals but by sets of clauses. This complicates the computation somewhat.

Let C_i be a set of clauses characterizing those states that are reachable by i operator applications. Similarly to distance computation, we consider for each operator and for each clause in C_i whether applying the operator may make the clause false. If it can, the clause could be false after i operator applications and therefore will not be in the clause set C_{i+1}.

For this basic step of invariant computation, whether an operator application may falsify a clause, we present two algorithms, first a simple one for a restricted class of operators, and then a more general for arbitrary operators.

Figure 3.3 gives an algorithm that tests whether applying an operator $o \in O$ in some state s may make a formula $l_1 \vee \cdots \vee l_n$ false assuming that $s \models \Delta \cup \{l_1 \vee \cdots \vee l_n\}$.

The algorithm performs a case analysis for every literal in the clause, testing in each case that the clause remains true: if a literal becomes false, either some other literal in the clause becomes true simultaneously or some other literal in the clause was true already and does not become false.

The algorithm is defined only for operators that have a precondition that is a conjunction of literals and an effect that is a conjunction of atomic effects (known as STRIPS operators for historical reasons). We give a similar algorithm for arbitrary operators later in Figure 3.4.

procedure simplepreserved(ϕ,Δ,o);
Now $\phi = l_1 \vee \cdots \vee l_n$ and $o = \langle l'_1 \wedge \cdots \wedge l'_{n'}, l''_1 \wedge \cdots \wedge l''_{n''} \rangle$ for some l_i, l'_j and l''_k;
if $\{\overline{l'''_1}, \cdots, \overline{l'''_m}\} \subseteq \{l'_1, \ldots, l'_{n'}\}$ for some $l'''_1 \vee \cdots \vee l'''_m \in \Delta$ **then return** true;
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ *(* Operator is not applicable. *)*

$\quad$ **for each** $l \in \{l_1, \ldots, l_n\}$ **do**
$\qquad$ **if** $\bar{l} \notin \{l''_1, \ldots, l''_{n''}\}$ **then goto** OK; $\qquad\qquad$ *(* Literal l cannot become false. *)*
$\qquad$ **for each** $l' \in \{l_1, \ldots, l_n\} \setminus \{l\}$ **do**
$\qquad\quad$ **if** $l' \in \{l''_1, \ldots, l''_{n''}\}$ **then goto** OK; $\qquad\qquad$ *(* Literal l' becomes true. *)*
$\qquad\quad$ **if** $l' \in \{l'_1, \ldots, l'_{n'}\}$ **or** $\overline{l'''_1} \vee \cdots \vee \overline{l'''_m} \vee l' \in \Delta$ for some $\{l'''_1, \ldots, l'''_m\} \subseteq \{l'_1, \ldots, l'_{n'}\}$,
$\qquad\quad$ **and** $\overline{l'} \notin \{l''_1, \ldots, l''_{n''}\}$
$\qquad\quad$ **then goto** OK; $\qquad\qquad\qquad$ *(* Literal l' was true and cannot become false. *)*
$\qquad$ **end do**
$\qquad$ **return** false; $\qquad\qquad\qquad$ *(* Truth of the clause could not be guaranteed. *)*
$\qquad$ OK:
$\quad$ **end do**
$\quad$ **return** true;

Figure 3.3: Algorithm that tests if applying o may falsify $l_1 \vee \cdots \vee l_n$ in a state satisfying Δ

Lemma 3.35 *Let Δ be a set of clauses, $\phi = l_1 \vee \cdots \vee l_n$ a clause, and o an operator of the form $\langle l'_1 \wedge \cdots \wedge l'_{n'}, l''_1 \wedge \cdots \wedge l''_{n''} \rangle$ where l'_j and l''_k are literals. If simplepreserved(ϕ,Δ,o) returns* true, *then $app_o(s) \models \phi$ for any state s such that $s \models \Delta \cup \{\phi\}$ and o is applicable in s. (It may under these conditions also return* false*).*

Proof: Assume s is a state such that $s \models l'_1 \wedge \cdots \wedge l'_{n'}$ and $s \models \Delta$ and $s \models \phi$ and $app_o(s) \not\models \phi$. We show that the procedure returns *false*.

Because $s \models \phi$ and $app_o(s) \not\models \phi$, there are some literals $\{l^f_1, \ldots, l^f_m\} \subseteq \{l_1, \ldots l_n\}$ such that $s \models l^f_1 \wedge \cdots \wedge l^f_m$ and $\{\overline{l^f_1}, \ldots, \overline{l^f_m}\} \subseteq \{l''_1, \ldots l''_{n''}\}$, that is, applying o makes them false, and the rest of the literals in ϕ were false and do not become true.

Choose any $l \in \{l^f_1, \ldots, l^f_m\}$. We show that when the outermost *for each* loop considers l the procedure will return *false*.

By assumption $\bar{l} \in \{l''_1, \ldots l''_{n''}\}$, and the condition of the first *if* inside the loop is not satisfied and the execution proceeds by iteration of the inner *for each* loop.

Let l' be any of the literals in ϕ except l.

Because ϕ is false in $app_o(s)$, $l' \notin \{l''_1, \ldots l''_{n''}\}$, and the condition of the first *if* statement is not satisfied.

If $l' \in \{l^f_1, \ldots, l^f_m\}$ then by assumption $\overline{l'} \in \{l''_1, \ldots, l''_{n''}\}$ and the condition of the second *if* statement is not satisfied.

If $l' \notin \{l^f_1, \ldots, l^f_m\}$ then by assumption $s \not\models l'$. Because the operator is applicable $s \models l'_1 \wedge \cdots \wedge l'_{n'}$, and hence $l' \notin \{l'_1, \ldots \wedge l'_{n'}\}$. Because s satisfies the precondition $l'_1 \wedge \cdots \wedge l'_{n'}$ and $s \models \Delta$, there is also no $\overline{l''} \vee l' \in \Delta$ for any $l'' \in \{l'_1, \ldots, l'_{n'}\}$. Hence also in this case the condition of the *if* statement is not satisfied.

Hence on none of the iterations of the inner *for each* loop is a *goto OK* executed, and as the loop exits, the procedure returns *false*. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

Figure 3.4 gives a similar algorithm for arbitrary operators. The structure of the algorithm is

procedure preserved(ϕ,Δ,o);
Now $\phi = l_1 \vee \cdots \vee l_n$ for some $l_1, \ldots, l_n$ and $o = \langle c, e \rangle$ for some c and e;
if $\Delta \models \neg c$ **then return** true;   *(* Operator is not applicable. *)*
for each $l \in \{l_1, \ldots, l_n\}$ **do**
  if $\Delta \wedge \{EPC_{\bar{l}}(e)\} \models \bot$ **then goto** OK;   *(* Literal l cannot become false. *)*
  for each $l' \in \{l_1, \ldots, l_n\} \backslash \{l\}$ **do**
    if $\Delta \cup \{EPC_{\bar{l}}(e), c\} \models EPC_{l'}(e)$ **then goto** OK;   *(* Literal l' becomes true. *)*
    if $\Delta \cup \{EPC_{\bar{l}}(e), c\} \models l'$ **and** $\Delta \cup \{EPC_{\bar{l}}(e), c\} \models \neg EPC_{\overline{l'}}(e)$ **then goto** OK;
                                     (* Literal l' was true and cannot become false. *)
  end do
  return false;   *(* Truth of the clause could not be guaranteed. *)*
  OK:
end do
return true;

Figure 3.4: Algorithm that tests if applying o may falsify $l_1 \vee \cdots \vee l_n$ in a state satisfying Δ

exactly the same, but the tests whether a certain literal becomes true or false or whether it was true before the operator was applied, are more complicated.

The algorithm is allowed to fail in one direction: it may sometimes return false *when* $l_1 \vee \cdots \vee l_n$ *actually is true after applying the operator. However, this is a necessary consequence of our requirement that our invariant computation takes only polynomial time.*

Lemma 3.36 *Let Δ be a set of clauses, $\phi = l_1 \vee \cdots \vee l_n$ a clause, and o an operator. If preserved(ϕ,Δ,o) returns* true*, then $app_o(s) \models \phi$ for any state s such that $s \models \Delta \cup \{\phi\}$ and o is applicable in s. (It may under these conditions also return* false*).*

Proof: $\square$

Figure 3.5 gives the algorithm for computing invariants consisting of at most n literals.

Theorem 3.37 *Let A be a set of state variables, I a state, O a set of operators, and $n \geq 1$ an integer.*

Then the procedure call invariants(A, I, O, n) returns a set C' of clauses so that for any sequence $o_1; \ldots, o_m$ of operators from O $app_{o_1;\ldots,o_m}(I) \models C'$.

Proof: Let C_0 be the value first applied to the variable C in the procedure *invariants*, and $C_1, C_2, \ldots$ be the values of the variable in the end of each iteration of the outermost *repeat* loop.

Induction hypothesis: for every $\phi \in C_i$, $app_{o_1;\ldots,o_i}(I) \models \phi$.

Base case $i = 0$: $app_\epsilon(I)$ for the empty sequence is by definition I itself, and by construction C_0 consists of only formulae that are true in the initial state.

Inductive case $i \geq 1$:

$\square$

The algorithm in Figure 3.4 does not run in polynomial time in the size of the problem instance because the logical consequence tests may take exponential time. To make the procedure run in polynomial time, we can again use an approximate logical consequence test, similar to the

procedure invariants(A, I, O, n);
$C := \{a \in A | I \models a\} \cup \{\neg a | a \in A, I \not\models a\}$;
repeat
  $C' := C$;
  for each $l_1 \vee \cdots \vee l_m \in C$ **do**   *(* Test every clause *)*
    for each $o \in O$ **do**   *(* with respect to every operator. *)*
      $N := \{l_1 \vee \cdots \vee l_m\}$;
      repeat
        $N' := N$;
        for each $l'_1 \vee \cdots \vee l'_{m'} \in N$ s.t. **not** preserved$(l'_1 \vee \cdots \vee l'_{m'}, C', o)$ **do**
          $N := N \backslash \{l'_1 \vee \cdots \vee l'_{m'}\}$;
          if $m' < n$ **then**   *(* Clause length within pre-defined limit. *)*
            begin
              $N := N \cup \{l'_1 \vee \cdots \vee l'_{m'} \vee a \mid a \in A\}$;
              $N := N \cup \{l'_1 \vee \cdots \vee l'_{m'} \vee \neg a \mid a \in A\}$;
            end
        end do
      until $N = N'$;   *(* N was not weakened further. *)*
      $C := (C \backslash \{l_1 \vee \cdots \vee l_m\}) \cup N$;
    end do
  end do
until $C = C'$;
return C;

Figure 3.5: Algorithm for computing a set of invariant clauses

procedure canbetruein(ϕ, D) used in Definition 3.16. The logical consequence test is allowed to fail in one direction without invalidating the invariant algorithm in Figure 3.5: *preserved* is allowed to return *false* also when the operator would not falsify ϕ, and hence logical consequence tests may be answered *no* even when the correct answer is *yes*.

The logical consequence tests have the form $\Delta \cup S \models \phi$. The logical consequence $\Delta \cup S \models \phi$ holds if and only if $\Delta \cup \{\bigwedge S \wedge \neg\phi\}$ is not satisfiable. A correct approximation is allowed to answer *satisfiable* even when the formula is unsatisfiable.

We present a polynomial time approximation of satisfiability tests for sets of formulae $\Delta \cup S$ in the case in which Δ consists of clauses of length at most 2. It is based on the definition of sets of literals litconseqs(ϕ, Δ) given below. The idea of litconseqs(ϕ, Δ) is that this set consists of (a subset of the) literals that must be true when ϕ and Δ are true, that is, that are logical consequences of ϕ and Δ. The one-sided error litconseqs(ϕ, Δ) is allowed to make and indeed does make is how disjunction $\vee$ is handled. if $\Delta \cup \{\phi\}$ is satisfiable, then litconseqs(ϕ, Δ) does not contain $\perp$ nor a and $\neg a$ for any $a \in A$.

$$
\begin{aligned}
\text{litconseqs}(\perp, \Delta) &= \{\perp\} \\
\text{litconseqs}(\top, \Delta) &= (\Delta \cap A) \cup (\Delta \cap \{\neg a | a \in A\}) \\
\text{litconseqs}(a, \Delta) &= \{a\} \cup \{l | \neg a \vee l \in \Delta\} \cup (\Delta \cap A) \cup (\Delta \cap \{\neg a | a \in A\}) \\
\text{litconseqs}(\neg a, \Delta) &= \{\neg a\} \cup \{l | a \vee l \in \Delta\} \cup (\Delta \cap A) \cup (\Delta \cap \{\neg a | a \in A\}) \\
\text{litconseqs}(\neg\neg\phi, \Delta) &= \text{litconseqs}(\phi, \Delta) \\
\text{litconseqs}(\phi \vee \psi, \Delta) &= \text{litconseqs}(\phi, \Delta) \cap \text{litconseqs}(\psi, \Delta) \\
\text{litconseqs}(\phi \wedge \psi, \Delta) &= \text{litconseqs}(\phi, \Delta) \cup \text{litconseqs}(\psi, \Delta) \\
\text{litconseqs}(\neg(\phi \vee \psi), \Delta) &= \text{litconseqs}(\neg\phi, \Delta) \cup \text{litconseqs}(\neg\psi, \Delta) \\
\text{litconseqs}(\neg(\phi \wedge \psi), \Delta) &= \text{litconseqs}(\neg\phi, \Delta) \cap \text{litconseqs}(\neg\psi, \Delta)
\end{aligned}
$$

The approximation fails because the satisfiability test is too simple. Consider litconseqs$((A \vee B) \wedge \neg(A \vee B), \emptyset)$ which is the empty set of literals because litconseqs$(A \vee B, \emptyset) = \emptyset$ and litconseqs$(\neg(A \vee B), \emptyset) = \emptyset$. This formula is unsatisfiable because it has the form $\phi \wedge \neg\phi$.

There are some simple ways of strengthening this approximation. For example, conjunction could be strengthened to

$$\text{litconseqs}(\phi \wedge \psi, \Delta) = \text{litconseqs}(\phi, \Delta \cup \text{litconseqs}(\psi, \Delta)) \cup \text{litconseqs}(\psi, \Delta \cup \text{litconseqs}(\phi, \Delta))$$

and further by computing more consequences for one of the conjuncts with the literals obtained from the other until no more literals are obtained.

The function litconseqs(ϕ, Δ) can also be used as a part of slightly more powerful (???) logical consequence tests as follows.

Define

$$
\begin{aligned}
\text{entailed}(\perp, D) &= \text{false} \\
\text{entailed}(\top, D) &= \text{true} \\
\text{entailed}(a, D) &= \text{true iff } a \in D \text{ (for state variables } a \in A) \\
\text{entailed}(\neg a, D) &= \text{true iff } \neg a \in D \text{ (for state variables } a \in A) \\
\text{entailed}(\neg\neg\phi, D) &= \text{entailed}(\phi, D) \\
\text{entailed}(\phi \vee \psi, D) &= \text{entailed}(\phi, D) \text{ or entailed}(\psi, D) \\
\text{entailed}(\phi \wedge \psi, D) &= \text{entailed}(\phi, D) \text{ and entailed}(\psi, D) \\
\text{entailed}(\neg(\phi \vee \psi), D) &= \text{entailed}(\neg\phi, D) \text{ and entailed}(\neg\psi, D) \\
\text{entailed}(\neg(\phi \wedge \psi), D) &= \text{entailed}(\neg\phi, D) \text{ or entailed}(\neg\psi, D)
\end{aligned}
$$

Notice that the definition of entailed(ϕ, D) is similar to canbetruein(ϕ, D) in Definition 3.16 except that literals a and $\neg a$ are handled differently: entailed(ϕ, D) is about logical consequences of D, that is formulae that are guaranteed to be true when D is true, while canbetruein(ϕ, D) is about ϕ being consistent with D.

Now if entailed$(\phi, \text{litconseqs}(\psi, \Delta))$ then $\Delta \cup \{\psi\} \models \phi$.

3.6.2 Applications in planning by regression and satisfiability

The first application is in planning in the propositional logic. It has been noticed that adding the 2-literal invariants to all time points reduces runtimes of algorithms that test satisfiability. Notice that invariants do not affect the set of models of a formula representing planning: any satisfying valuation of the original formula also satisfies the invariants, because the values of propositions describing the values of state variables at any time point corresponds to a state that is reachable from the initial state, and hence this valuation also satisfies any invariant.

The second application is in planning by regression. Consider the blocks world with the goal A-ON-B $\wedge$ B-ON-C. Now we can regress with the operator that moves B onto C from the table, obtaining the new goal A-ON-B $\wedge$ B-CLEAR $\wedge$ C-CLEAR $\wedge$ B-ON-TABLE. Clearly, this does not correspond to an intended blocks world state because A-ON-B is incompatible with B-CLEAR, and indeed, $\neg$A-ON-B $\vee$ $\neg$B-CLEAR is an invariant for the blocks world. Any regression step that leads to a goal that is incompatible with the invariants can be ignored, because that goal does not represent any of the states that are reachable from the initial state, and hence no plan can reach the goal in question.

Another application of invariants, and the intermediate sets C_i produced by our invariant algorithm, is improving the distance estimation in Section 3.4. Using v_i for testing whether an operator precondition, for example $a \wedge b$, has distance i from the initial state, the distances of a and b are used separately. But even when it is possible to reach both a and b with i operator applications, it might still not be possible to reach them both simultaneously with i operator applications. For example, for $i = 1$ and an initial state in which both a and b are false, there might be no single operator that makes them both true, but two operators, each of which makes only one of them true. If $\neg a \vee \neg b \in C_i$, we know that after i operator applications one of a or b must still be false, and then we know that the operator in question is not applicable at time point i. Therefore the invariants and the sets C_i produced during the invariant computation can improve the distance estimates.

3.7 Planning with symbolic representations of sets of states

A complementary approach to planning for planning problems represented as formulae in the propositional logic uses the formulae as a data structure. As discussed in Section 2.3.3 formulae directly provide a representation of sets of states, and in this section we show how operations on transition relations have a counterpart as operations on formulae that represent transition relations.

This yields a further planning algorithm for deterministic planning, typically implemented by means of BDDs. The algorithm in Section 3.7.3 will later be generalized to different types of nondeterministic planning.

Table 3.1 outlines a number of connections between operations on vectors and matrices, on propositional formulae, and on sets and relations.

Computing the product of two matrices that are represented as propositional formulae is based on the *existential abstraction* operation $\exists p.\phi = \phi[\top/p] \vee \phi[\bot/p]$ that takes a formula ϕ and a

matrices	formulas	sets of states
vector $V_{1 \times n}$	formula over A	set of states
matrix $M_{n \times n}$	formula over $A \cup A'$	transition relation
$M_{n \times n} \times N_{n \times n}$	$\exists A'.(\phi(A, A') \wedge \psi(A', A''))$	sequential composition
$S_{1 \times n} \times M_{n \times n}$	$\exists A.(\phi(A) \wedge \psi(A, A'))$	successor states of S
$S_{1 \times n} + S'_{1 \times n}$	$\phi \vee \psi$	set union
	$\phi \wedge \psi$	set intersection

Table 3.1: Correspondence between matrix operations, Boolean operations as well as set-theoretic and relational operations

proposition p and produces a formula ϕ' without occurrences of p.

Let ϕ be a formula over $A \cup A'$ and ψ be a formula over $A' \cup A''$. Now matrix product of matrices corresponding to ϕ and ψ' is

$$\exists A'.\phi \wedge \psi.$$

Example 3.38 Let $\phi = A \leftrightarrow \neg A'$ and $\psi = A' \leftrightarrow A''$ represent two actions, reversing the truth-value of A and doing nothing. The sequential composition of these actions is

$$\begin{aligned}
\exists A'.\phi \wedge \psi &= ((A \leftrightarrow \neg\top) \wedge (\top \leftrightarrow A'')) \vee ((A \leftrightarrow \neg\bot) \wedge (\bot \leftrightarrow A'')) \\
&\equiv ((A \leftrightarrow \bot) \wedge (\top \leftrightarrow A'')) \vee ((A \leftrightarrow \top) \wedge (\bot \leftrightarrow A'')) \\
&\equiv A \leftrightarrow \neg A''
\end{aligned}$$

∎

Consider the representation of planning as satisfiability in the propositional discussed in Section 3.5.3.

$$\iota^0 \wedge \underbrace{\mathcal{R}_1(A^0, A^1) \wedge \mathcal{R}_1(A^1, A^2) \wedge \cdots \wedge \mathcal{R}_1(A^{n-1}, A^n)} \wedge G^n$$

The conjunction of formulae the $\mathcal{R}_1(A^i, A^{i+1})$ representing the transition relation corresponds to the computation of the n-fold product of the corresponding adjacency matrices. Further, when the first factor in the product is the vector describing the initial state, we have the computation of the set of states reachable in n steps.

$$\underbrace{\underbrace{\iota^0 \times (\underbrace{\mathcal{R}_1(A^0, A^1)} \times \mathcal{R}_1(A^1, A^2) \times \cdots \times \mathcal{R}_1(A^{n-1}, A^n))}}$$

Taking the intersection of this set with the set of goal states tells us whether there is a plan of length n.

In the following we discuss how this idea can be turned into a planning algorithm, in which the n-fold product of the initial state vector with the adjacency matrices is computed step by step, yielding vectors describing the sets of states reachable in $i \in \{0, \dots, n\}$ operator applications.

3.7.1 Operations on transition relations expressed as formulae

The most basic operation is the computation of *the image* of a set of states with respect to a transition relation.

$$img_R(S) = \{s' | s \in S, \langle s, s' \rangle \in R\}$$

This is the set of states that can be reached from S by transition relation R. When sets of states and transition relation are represented as propositional formulae, the image computation can be performed by the existential abstraction and renaming operations as follows.

$$img_{\mathcal{R}(A,A')}(\phi) = (\exists A.(\phi \land \mathcal{R}(A, A')))[p_1/p_1', \dots, p_n/p_n']$$

Similarly we can compute the product of two matrices that are represented as formulae $\mathcal{R}(A, A')$ and $\mathcal{Q}(A', A'')$ by using existential abstraction.

$$\mathcal{R}(A, A') \cdot \mathcal{Q}(A', A'') = \exists A'.(\mathcal{R}(A, A') \land \mathcal{Q}(A', A''))$$

The resulting formula is over state variables A and A'', from which a formula on A and A' is obtained by renaming A'' to A'.

Plan search can also be performed starting from the goal states, like done with all the algorithms in Chapter 4. In this case we must compute sets of states from which any of the states in a given set can be reached by one step. This is represented as the computation of *the preimage* of a set of states with respect to a transition relation.[2]

$$wpreimg_R(S) = \{s | s' \in S, \langle s, s' \rangle \in R\}$$

This is the set of states from which a state in S is reached by the transition relation R. The corresponding computation in terms of formulae is as follows. Here ϕ is a formula over A, and first it has to be renamed to a formula over A'.

$$wpreimg_{\mathcal{R}(A,A')}(\phi) = \exists A'.(\phi[p_1'/p_1, \dots, p_n'/p_n] \land \mathcal{R}(A, A')) \tag{3.3}$$

Notice that when the relation $\mathcal{R}(A, A')$ corresponding to an operator o has been represented as discussed in Section 3.5.2, the Formula 3.3 for $wpreimg_{\mathcal{R}(A,A')}(\phi)$ is logically equivalent to the regression $regr_o(\phi)$ as given in Definition 3.6.

Example 3.39 Consider the formula $A \land B$ that is regressed with the operator $o = \langle C, A \land (A \triangleright B) \rangle$. Now we have

$$regr_o(\phi) = C \land (\top \land (B \lor A)) \equiv C \land (B \lor A).$$

The transition relation of o is represented by the formula

$$\tau = C \land A' \land ((B \lor A) \leftrightarrow B') \land (C \leftrightarrow C').$$

The preimage of $A \lor B$ with respect to o is represented by

$$
\begin{aligned}
\exists A'B'C'.((A' \land B') \land \tau) &\equiv \exists A'B'C'.((A' \land B') \land C \land A' \land ((B \lor A) \leftrightarrow B') \land (C \leftrightarrow C')) \\
&\equiv \exists A'B'C'.(A' \land B' \land C \land (B \lor A) \land C') \\
&\equiv \exists B'C'.(B' \land C \land (B \lor A) \land C') \\
&\equiv \exists C'.(C \land (B \lor A) \land C') \\
&\equiv C \land (B \lor A)
\end{aligned}
$$

$\blacksquare$

---

[2]This is often called the *weak preimage* to contrast it with the strong preimage operation defined in Section 4.3.

procedure planfwd(I,O,G)
$i := 0$;
$D_0 := \{I\}$;
while $G \cap D_i = \emptyset$ **and** $(i = 0$ **or** $D_{i-1} \neq D_i)$ **do**
  $i := i + 1$;
  $D_i := D_{i-1} \cup \bigcup_{o \in O} img_o(D_{i-1})$;   *(* Possible successors of states in D_{i-1} *)*
end
if $G \cap D_i = \emptyset$ **then** terminate;   *(* There is no plan. *)*
$S := G \cap D_i$;
for $j :=$ i-1 **to** 0 **do**   *(* Output plan, last operator first. *)*
  choose $o \in O$ such that $wpreimg_o(S) \cap D_j \neq \emptyset$;
  output o;
  $S := wpreimg_o(S) \cap D_j$;
end

Figure 3.6: Algorithm for deterministic planning (forward, in terms of sets)

As we will see later, computation of preimages is applicable to all kinds of operators, not only deterministic ones as required by our definition of regression, whereas defining regression for arbitrary operators is more difficult (we will give a definition of regression only for a subclass of nondeterministic operators.)

Hence our definition of regression can be viewed as a specialized method for computing preimage of formulae with respect to a transition relation corresponding to a deterministic operator. The main advantage of regression is that no existential abstraction is needed.

Notice that defining progression for arbitrary formulae (sets of states) seems to require existential abstraction. A simple syntactic definition of progression similar to that of regression does not seem to be possible because the value of state variable in a given state cannot be represented in terms of the values of the state variables in the successor state. This is because of the asymmetry of deterministic actions: the current state and an operator determine the successor state uniquely, but the successor state and the operator do not determine the current state uniquely. In other words, the changes that take place are a function of the current state, but not a function of the successor state.

3.7.2 A forward planning algorithm

The algorithm in Figure 3.6 has two phases: the computation of distance from the initial state to every reachable state, and the extraction of a plan. The set D_0 consists of the initial state, the set D_1 of those states that can be reached from the initial state by one operator, and so on.

We can express the same algorithm in terms of formulae in the propositional logic, see Figure 3.7. The plan extraction proceeds by identifying the operators in the backwards direction starting from the last one.

In the figure we give two variants of the algorithm, first expressed in terms of set-theoretic operations on sets of states and transition relations, and then expressed in terms of the propositional formulae.

Notice that in the first version of the algorithm D_i is computed as the union of D_{i-1} (reachability by $i - 1$ steps or less) and the images of D_{i-1} with respect to all of the operators, and hence D_i

procedure planfwd(I,$\mathcal{R}_1(A, A')$,G)
$i := 0$;
$D_0 := I$;
while $D_i \models \neg G$ **and** ($i = 0$ **or** $\not\models D_{i-1} \leftrightarrow D_i$) **do**
  $i := i + 1$;
  $D_i := (\exists A.(D_{i-1} \wedge \mathcal{R}_1(A, A')))[p'_1/p_1, \ldots, p'_n/p_n]$; *(* Possible predecessors of states in D_{i-1} *)*
end
if $D_i \models \neg G$ **then** terminate;   *(* There is no plan. *)*
$S := G \wedge D_i$;
for $j :=$ i-1 **to** 0 **do**   *(* Output plan, last operator first. *)*
  choose $o \in O$ such that $D_j \not\models \neg wpreimg_{\tau_o}(S)$;
  output o;
  $S := wpreimg_{\tau_o}(S) \wedge D_j$;
end

Figure 3.7: Algorithm for deterministic planning (forward, in terms of formulae)

represents reachability by i steps or less. In the second version the transition relation $\mathcal{R}_1(A, A')$ encodes reachability by 0 or 1 steps, so we directly obtain reachability by i steps or less, without having to take union ($\vee$) with D_{i-1}.

Theorem 3.40 *Let a state s be in $D_i \backslash D_{i-1}$. Then there is a plan that reaches s from the initial state by i operator applications.*

Proof: $\square$

3.7.3 A backward planning algorithm

The second algorithm computes the distances to the goal states. This computation proceeds by preimage computation starting from the goal states, so D_0 consists of the goal states, D_1 the states with distance 1 to the goal states, and so on. The algorithm is given in Figure 3.8.

We can express the same algorithm in terms of formulae in the propositional logic, see Figure 3.9.

Theorem 3.41 *Let a state s be in $D_i \backslash D_{i-1}$. Then there is a plan that reaches from s a goal state by i operator applications.*

Proof: $\square$

3.8 Computational complexity

In this section we discuss the computational complexity of the main decision problems related to deterministic planning.

The plan existence problem of deterministic planning is PSPACE-complete. The result was proved by Bylander [1994]. He proved the hardness part by giving a simulation of deterministic

procedure planbwd(I,O,G)
$D_0 := G$;
$i := 0$;
while $I \notin D_i$ **and** ($i = 0$ **or** $D_{i-1} \neq D_i$) **do**
  $i := i + 1$;
  $D_i := D_{i-1} \cup \bigcup_{o \in O} wpreimg_o(D_{i-1})$;
end
if $I \notin D_i$ **then** terminate;   *(* There is no plan. *)*
$s := I$;
for $j := i - 1$ **to** 0 **do**   *(* Output plan, first operator first. *)*
  choose $o \in O$ such that $app_o(s) \in D_j$;
  output o;
  $s := app_o(s)$;
end

Figure 3.8: Algorithm for deterministic planning (backward, in terms of states)

procedure planbwd(I,$\mathcal{R}_1(A, A')$,G)
$D_0 := G$;
$i := 0$;
while $I \not\models D_i$ **and** ($i = 0$ **or** $\not\models D_{i-1} \leftrightarrow D_i$) **do**
  $i := i + 1$;
  $D_i := \exists A'.(\mathcal{R}_1(A, A') \wedge (D_{i-1}[p'_1/p_1, \ldots, p'_n/p_n]))$;
end

Figure 3.9: Algorithm for deterministic planning (backward, in terms of formulae)

polynomial-space Turing machines, and the membership part by giving an algorithm that solves the problem in polynomial space. We later generalize his Turing machine simulation to alternating Turing machines to obtain an EXP-hardness proof for nondeterministic planning with full observability in Theorem 4.42.

Theorem 3.42 *The problem of testing the existence of a plan is PSPACE-hard.*

Proof: Let $\langle \Sigma, Q, \delta, q_0, g \rangle$ be any deterministic Turing machine with a polynomial space bound $p(x)$. Let σ be an input string of length n.

We construct a problem instance in deterministic planning with for simulating the Turing machine. The problem instance has a size that is polynomial in the size of the description of the Turing machine and the input string.

The set A of state variables in the problem instance consists of

1. $q \in Q$ for denoting the internal states of the TM,

2. s_i for every symbol $s \in \Sigma \cup \{|, \Box\}$ and tape cell $i \in \{0, \ldots, p(n)\}$, and

3. h_i for the positions of the R/W head $i \in \{0, \ldots, p(n) + 1\}$.

The initial state of the problem instance represents the initial configuration of the TM. The initial state I is as follows.

1. $I(q_0) = 1$

2. $I(q) = 0$ for all $q \in Q \backslash \{q_0\}$.

3. $I(s_i) = 1$ if and only if ith input symbol is $s \in \Sigma$, for all $i \in \{1, \ldots, n\}$.

4. $I(s_i) = 0$ for all $s \in \Sigma$ and $i \in \{0, n+1, n+2, \ldots, p(n)\}$.

5. $I(\Box_i) = 1$ for all $i \in \{n+1, \ldots, p(n)\}$.

6. $I(\Box_i) = 0$ for all $i \in \{0, \ldots, n\}$.

7. $I(|_0) = 1$

8. $I(|_i) = 0$ for all $n \in \{1, \ldots, p(n)\}$

9. $I(h_1) = 1$

10. $I(h_i) = 0$ for all $i \in \{0, 2, 3, 4, \ldots, p(n) + 1\}$

The goal is the following formula.

$$G = \bigvee \{q \in Q | g(q) = \text{accept}\}$$

To define the operators, we first define effects corresponding to all possible transitions.

For all $\langle s, q \rangle \in (\Sigma \cup \{|, \Box\}) \times Q$, $i \in \{0, \ldots, p(n)\}$ and $\langle s', q', m \rangle \in (\Sigma \cup \{|\}) \times Q \times \{L, N, R\}$ define the effect $\tau_{s,q,i}(s', q', m)$ as $\alpha \wedge \kappa \wedge \theta$ where the effects α, κ and θ are defined as follows.

The effect α describes what happens to the tape symbol under the R/W head. If $s = s'$ then $\alpha = \top$ as nothing on the tape changes. Otherwise, $\alpha = \neg s_i \wedge s'_i$ to denote that the new symbol in the ith tape cell is s' and not s.

procedure reach(O,s,s',m)
if $m = 0$ **then**   *(* Plans of length 0 and 1 *)*
  if s = s' **or** there is $o \in O$ such that $s' = app_o(s)$ **then return** true
  else return false
else
  begin   *(* Longer plans *)*
    for all states s'' **do**   *(* Iteration over intermediate states *)*
      if reach(O,s,s'',$m-1$) **and** reach(O,s'',s',$m-1$) **then return** true
    end
    return false;
  end

Figure 3.10: Algorithm for testing plan existence in polynomial space

The effect κ describes the change to the internal state of the TM. Again, either the state changes or does not, so $\kappa = \neg q \wedge q'$ if $q \neq q'$ and $\top$ otherwise. We define $\kappa = \neg q$ when $i = p(n)$ and $m = R$ so that when the space bound gets violated, no accepting state can be reached.

The effect θ describes the movement of the R/W head. Either there is movement to the left, no movement, or movement to the right. Hence

$$\theta \quad = \quad \begin{cases} \neg h_i \wedge h_{i-1} & \text{if } m = L \\ \top & \text{if } m = N \\ \neg h_i \wedge h_{i+1} & \text{if } m = R \end{cases}$$

By definition of TMs, movement at the left end of the tape is always to the right. Similarly, we have state variable for R/W head position $p(n) + 1$ and moving to that position is possible, but no transitions from that position are possible, as the space bound has been violated.

Now, these effects that represent possible transitions are used in the operators that simulate the Turing machine. Let $\langle s, q \rangle \in (\Sigma \cup \{|, \square\}) \times Q$, $i \in \{0, \ldots, p(n)\}$ and $\delta(s, q) = \{\langle s', q', m \rangle\}$. If $g(q) = \exists$, then define the operator

$$o_{s,q,i} = \langle h_i \wedge s_i \wedge q, \tau_{s,q,i}(s', q', m) \rangle.$$

We claim that the problem instance has a plan if and only if the Turing machine accepts without violating the space bound.

If the Turing machine violates the space bound, the state variable $h_{p(n)+1}$ becomes true and an accepting state cannot be reached because no further operator will be applicable.

So, because all deterministic Turing machines with a polynomial space bound can be in polynomial time translated to a planning problem, all decision problems in PSPACE are polynomial time many-one reducible to deterministic planning, and the plan existence problem is PSPACE-hard. $\square$

Theorem 3.43 *The problem of testing the existence of a plan is in PSPACE.*

Proof: A recursive algorithm for testing m-step reachability between two states with $\log m$ memory consumption is given in Figure 3.10.

We show that when the algorithm is called with the number $n = |A|$ of state variables as the last argument, it consumes a polynomial amount of memory in n. The recursion depth is n. At the

recursive calls memory is needed for storing the intermediate states s'. The memory needed for this is polynomial in n. Hence at any point of time the space consumption is $\mathcal{O}(m^2)$.

A problem instance $\langle A, I, O, G \rangle$ with $n = |A|$ state variables has a plan if and only if reach(O,I,s',n) returns *true* for some s' such that $s' \models G$. Iteration over all states s' can be performed in polynomial space and testing $s' \models G$ can be performed in polynomial time in the size of G. Hence the whole memory consumption is polynomial. $\qquad \square$

Part of the high complexity of planning is due to the fact that plans can be exponentially long. If a polynomial upper bound for plan length exists, testing the existence of plans is still intractable but much easier.

Theorem 3.44 *The problem of testing the existence of plans having a length bounded by some polynomial is NP-hard.*

Proof: We reduce the satisfiability problem of the classical propositional logic to the plan existence problem. The length of the plans, whenever they exist, is bounded by the number of propositional variables and hence is polynomial.

Let ϕ be a formula over the propositional variables in A. Let $N = \langle A, \{(a, 0)|a \in A\}, O, \phi \rangle$ where $O = \{\langle \top, a \rangle | a \in A\}$ We show that the problem instance N has a plan if and only if the formula ϕ is satisfiable.

Assume $\phi \in SAT$, that is, there is a valuation $v : A \rightarrow \{0, 1\}$ such that $v \models \phi$. Now take the operators $\{\langle \top, a \rangle | v \models a, a \in A\}$ in any order: these operators form a plan that reach the state v that satisfies ϕ.

Assume N has a plan $o_1, \ldots, o_m$. The valuation $v = \{(a, 1) | (\top, a) \in \{o_1, \ldots, o_m\}\} \cup \{(a, 0) | a \in A, (\top, a) \notin \{o_1, \ldots, o_m\}\}$ of A is the terminal state of the plan and satisfies ϕ. $\qquad \square$

Theorem 3.45 *The problem of testing the existence of plan having a length bounded by some polynomial is in NP.*

Proof: Let $p(m)$ be a polynomial. We give a nondeterministic algorithm that runs in polynomial time and determines whether a plan of length $p(m)$ exists.

Let $N = \langle A, I, O, G \rangle$ be a problem instance.

1. Nondeterministically guess a sequence of $l \leq p(m)$ operators $o_1, \ldots, o_l$ from the set O. Because l is bounded by the polynomial $p(m)$, the time consumption $\mathcal{O}(p(m))$ is polynomial in the size of N.

2. Compute $s = app_{o_l}(app_{o_{l-1}}(\cdots app_{o_2}(app_{o_1}(I)) \cdots))$. This takes polynomial time in the size of the operators and the number of state variables.

3. Test $s \models G$. This takes polynomial time in the size of the operators and the number of state variables.

This nondeterministic algorithm correctly determines whether a plan of length at most $p(m)$ exists and it runs in nondeterministic polynomial time. Hence the problem is in NP. $\qquad \square$

These theorems show the NP-completeness of the plan existence problem for polynomial-length plans.

3.9 Literature

The idea of progression and regression in planning is old [Rosenschein, 1981]. Our definition of regression in Section 3.2.2 is related to the weakest precondition predicates for program synthesis [de Bakker and de Roever, 1972; Dijkstra, 1976]. Planning researchers have earlier used regression only for a very restricted type of operators without conditional effects.

There has recently been a lot of interest in using general-purpose search algorithms with progression and heuristics that estimate distances between states. Our distance estimation in Section 3.4 generalizes the additive heuristic by Bonet and Geffner [2001] by handling the truth-values symmetrically and by being applicable to a more type of operators with arbitrary preconditions and conditional effects. Other distance estimates with a flavor that is similar to Bonet and Geffner's exist [Haslum and Geffner, 2000; Hoffmann and Nebel, 2001].

Techniques for speeding up heuristic state-space planners include symmetry reduction [Starke, 1991; Emerson and Sistla, 1996] and partial-order reduction [Godefroid, 1991; Valmari, 1991; Alur *et al.*, 1997], both originally introduced outside planning in the context of reachability analysis and model-checking. Both of these techniques address the main problem in heuristic state-space search, high branching factor (number of applicable operators) and high number of states. Both techniques help in reducing the number of states to be traversed when searching for a plan.

The use of algorithms for the satisfiability problem of the classical propositional logic in planning was pioneered by Kautz and Selman, originally as a way of testing satisfiability algorithms, and later shown to be more efficient than other planning algorithms at that time [Kautz and Selman, 1992; 1996]. In addition to Kautz and Selman [1996], parallel plans were used by Blum and Furst in their Graphplan planner [Blum and Furst, 1997]. Parallelism in this context serves the same purpose as partial-order reduction [Godefroid, 1991; Valmari, 1991], namely to avoid considering all orderings of a number of independent actions and hence reduce the amount of search. The notion of parallel plans considered in this lecture is not the only possible one [Rintanen *et al.*, 2004].

The algorithm for invariant computation was originally presented for simple operators without conditional effects [Rintanen, 1998]. The computation parallels the construction of planning graphs in the Graphplan algorithm [Blum and Furst, 1997], and it would seem to us that the notion of planning graph emerged when Blum and Furst noticed that the intermediate stages of the invariant computation are useful for backward search algorithms: if a depth-bound of n is imposed on the search tree, then formulae obtained by m regression steps (suffixes of length m of possible plans) that do not satisfy clauses R_{n-m} cannot lead to a plan, and the search tree can be pruned.

Even though a lot of contemporary planning research uses Graphplan's planning graphs [Blum and Furst, 1997] for various purposes, we have not discussed them in more detail for several reasons. First, the graph character of planning graphs becomes inconvenient when preconditions are arbitrary formulae, not just conjunctions of state variables, and effects may be conditional. As a result, the basic construction steps of planning graphs become unintuitive. Second, even when the operators have the simple form, the practically and theoretically important properties of planning graphs are not graph-theoretic. We can equivalently and just as intuitively represent the contents of planning graphs as sequences of literals and 2-literal clauses, as we have done for instance in Section 3.6. So it seems that the graph representation does not provide advantages over more conventional logic based and set based representations.

The algorithms presented in this section cannot in general be ordered in terms of efficiency. The general-purpose search algorithms with distance heuristics are often very effective in solving

big problem instances with a suitable structure. Sometimes this entails better runtimes than in the SAT/CSP approach because of the high overheads with handling big formulae or constraint nets in the latter. Similarly, there are problems that are quickly solved by the SAT/CSP approach but on which the distance estimation fails and the heuristic search algorithms are not able to find plans quickly.

The main complexity result of the chapter, the PSPACE-completeness of the plan existence problem, is due to Bylander [1994]. Essentially the same result for other kinds of succinct representations of graphs had been established earlier by Lozano and Balcazar [1990].

Any computational problem just NP-hard – not to mention PSPACE-hard – is usually considered to be too difficult to be solved in any but the simplest cases. Because planning even in the deterministic case is PSPACE-hard, there has been interest in finding useful special cases in which it can be guaranteed that the worst-case complexity does not show up. Syntactic restrictions leading to polynomial time planning have been investigated by several researchers [Bylander, 1994; Bäckström and Nebel, 1995], but the restrictions are so strict that very few or no interesting problems can be represented.

The computational complexity of planning with schematic operators has also been analyzed. Schematic operators increase the conciseness of the representations of some problem instances exponentially, and lift the worst-case complexity accordingly. For example, deterministic planning with schematic operators is EXPSPACE-complete [Erol *et al.*, 1995]. If function symbols are allowed, encoding arbitrary Turing machines becomes possible, and the plan existence problem consequently becomes undecidable [Erol *et al.*, 1995].

3.10 Exercises

3.1 Show that regression for goals G that are sets (conjunctions) of state variables and operators with preconditions p that are sets (conjunctions) of state variables and effects that consist of an add list a (a set of state variables that become true) and a delete list d (a set of state variables that become false) can equivalently be defined as $(G \backslash a) \cup p$ when $d \cap G = \emptyset$.

3.2 Show that the problem in Lemma 3.9 is in NP and therefore NP-complete.

3.3 Satisfiability testing in the propositional logic is tractable in some special cases, like for sets of clauses with at most 2 literals in each, and for Horn clauses, that is sets of clauses with at most one positive literal in each clause.

Can you identify special cases in which existence of an n-step plan can be determined in polynomial time (in n and the size of the problem instance), because the corresponding formula transformed to CNF is a set of 2-literal clauses or a set of Horn clauses?